POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica - Computer Networks and Cloud Computing

Tesi di Laurea Magistrale

# Ottimizzazione dei costi in un cluster AWS tramite l'utilizzo di Knative


**Relatore**
prof. Fulvio Risso

**Laureando**
Marco Barbaro

**Corley srl**
Walter Dal Mut

Marzo 2022

# Ringraziamenti

Alla mia famiglia, in particolare ai miei genitori, che mi hanno sempre supportato nel raggiungere questo traguardo e di avermi spronato a continuare per raggiungere l'obiettivo. Ringrazio i miei amici per tutti i momenti di svago, come il "torneone", che mi hanno permesso di staccare dal mondo dell'università. Infine, un ringraziamento speciale ad Alessia, la mia ragazza, per avermi accompagnato durante tutti questi anni di studi e aver creduto sempre in me. Grazie per essere stata una costante nella mia vita, di avermi incoraggiato nei momenti bui e gioito insieme a me dei traguardi raggiunti.

# Sommario

In questi ultimi anni, sempre più aziende cercano di migrare da una tecnologia on-premise ad una infrastruttura cloud. Tale scelta è dovuta principalmente ai costi, è risaputo infatti che, spostare i propri workload sul cloud consente un notevole risparmio in termini di energia, di hardware e di gestione. L'obiettivo di questo lavoro permette un ulteriore passo verso il risparmio dei costi all'interno di un cloud provider. Numerose aziende dispongono di svariati servizi continuamente attivi all'interno del proprio cluster i quali vengono utilizzati poche volte durante l'intero arco di una giornata o ,addirittura, di un intero anno, occupando inutilmente risorse hardware e facendo lievitare i costi di gestione. A fronte di ciò, durante questi anni si è incominciato a parlare di architetture serverless e "Function as a Service". Queste ultime permettono di focalizzarsi sulla scrittura del codice dell'applicazione, senza doversi occupare della sua configurazione, e sulla messa in campo all'interno di un cluster permettendo di fatto un abbassamento dei costi. Oltre a soluzioni commerciali disponibili all'interno dei vari cloud provider, se ne sono affiancate altre open-source come ad esempio Knative. Tale framework si colloca on-top dell'infrastruttura di Kubernetes, estendendendo così le sue risorse e le sue funzionalità offerte. La sua peculiarità è la possibilità di implementazione del codice attraverso delle immagini a container, create da Docker per poi essere eseguite solo quando necessario, garantendo un notevole risparmio di risorse hardware all'interno di un cluster e abbassando i costi verso le aziende. Per raggiungere l'obiettivo di questo lavoro, si è voluto analizzare Knative e i servizi offerti da Amazon Web Services, cloud provider di riferimento in questa tesi. Al fine di analizzare queste tecnologie e per un confronto diretto di performance e costi si è reso necessario lo sviluppo da un lato un'applicazione monolitica, dall'altro di una equivalente, adempiente agli stessi compiti, ma utilizzando Knative. Successivamente, l'applicazione è stata ricostruita in una terza versione per poter essere integrata con i servizi forniti da AWS, in particolare AWS Lambda e Aurora serverless.
Ci si aspetta, come punto di arrivo, come sarà confermato dall'analisi suddetta, che i costi di una applicazione basata su Kubernetes risultino più alti rispetto ad una implementazione equivalente con Knative e AWS, in particolare a fronte di un numero maggiore di applicazioni all'interno del cluster.

# Indice

# Elenco delle tabelle

# Elenco delle figure

# Elenco dei listati

# Capitolo 1

# Introduzione

Nel corso di questi anni, lo scenario delle tecnologie informatiche si è evoluto in modo sostanziale attraversando diverse fasi. A partire da uno scenario come il *grid computing* si è arrivati a soluzioni moderne come il *cloud computing* che permette di abbattere gli elevati costi di manutenzione e acquisto hardware che un un datacenter dovrebbe avere.

Il *grid computing* consiste in una infrastruttura di calcolo distribuita con il compito di elaborare problemi complessi mediante l'uso di una grande quantità di risorse hardware e di rete. Il problema viene suddiviso in compiti più piccoli denominati *grid* che verranno successivamente eseguiti dai nodi del cluster. Questo metodo fa si che che il cluster si comporti come un supercomputer virtuale che permette l'accesso ai vari nodi geograficamente distribuite e fornendo una singola interfaccia di accesso [1].

Il *cloud computing* è un modello la cui peculiare caratteristica è quella di permettere l'accesso su network a un pool condiviso di risorse di elaborazione configurabili ( quali reti, server, archiviazione, applicazioni e servizi), in maniera ubiqua, conveniente e on-demand, che possono essere rapidamente forniti e rilasciati sotto forma di architettura distribuita, con il minimo sforzo di gestione o interazione con i fornitori di servizi [2].I vantaggi sull'uso del cloud computing sono numerevoli:

- **elasticità**: si ha la possibilità di scalare in alto o in basso l'utilizzo di risorse hardware a disposizione dell'utente in base a ciò che l'applicazione o il problema richiede.

- **economia di scala**: ottenuta grazie a numerose compagnie, come *Amazon* e *Google*, le quali forniscono l'accesso ai loro datacenter a numerosi utenti contemporaneamente, garantendo di fatto un'ottimizzazione delle risorse e conseguentemente un abbattimento dei costi che si riversa sul prezzo finale proposto all'utente.

- **Aggiornamenti**: gli aggiornamenti dei vari software vengono eseguiti istantaneamente in modo automatico, senza che l'utente debba occuparsene direttamente.

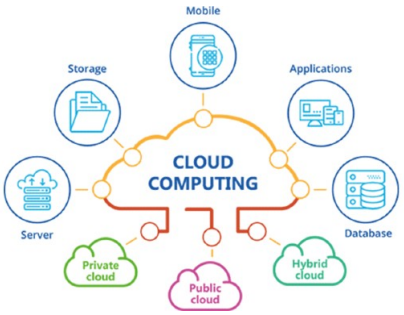



Figura 1.1: Architettura cloud computing

## 1.1 Modelli di cloud Computing

Nell'ambito del cloud computing esistono tre modelli principali, che si differenziano per il diverso livello di controllo e gestione delle risorse:

- **Infrastructure as a service**: consiste nell'utilizzo di un'infrastruttura di calcolo fornita e gestita attraverso la rete internet. Ciò che viene fornito all'utilizzatore consiste in una macchina virtualizzata, permettendogli di pagare solo per le risorse effettivamente utilizzate. Il provider mette a disposizione i propri server, il che consente di avere una soluzione scalabile, con le risorse che possono essere aumentate (o diminuite) in qualunque momento.

- **Platform as a service**: fornisce una piattaforma che permette agli utenti di sviluppare, eseguire e gestire le proprie applicazioni senza avere la complessità di costruire e mantenere l'infrastruttura tipicamente associata allo sviluppo di un'applicazione. In questo modello quindi si nascondono i "dettagli" come il sistema operativo e l'infrastruttura di rete.

- **Software as a service**: consiste nell'utilizzo di applicazioni complete, basate su cloud e residenti in esso, accessibili via internet. Gli utenti hanno il completo accesso alle funzionalità del servizio senza dover scrivere nessuna linea di codice. Di contro, la possibilità di poter customizzare il servizio a proprio piacimento risulta estremamente difficile.

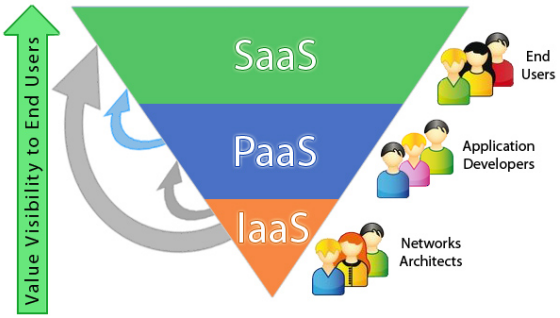



Figura 1.2: Modelli cloud computing

### 1.1.1 Serverless computing e Function as a service

La crescente popolarità del cloud computing ha portato a una nuova evoluzione dei modelli descritti precedentemente implementando nuove tecnologie, delle quali due esempi sono *Function-as-a-Service* e *serverless computing*, che mirano entrambe a:

- Fornire piattaforme cloud convenienti.
- Eliminare la necessità di gestione dell'infrastruttura.

Talvolta questi due termini sono usati in modo intercambiabile. Tuttavia, sono due tecnologie diverse con alcune differenze significative.

Il *serverless computing* consiste in un modello di elaborazione in cui l'orchestrazione dell'infrastruttura è gestita dai fornitori dei servizi, facendo in modo che ci si concentri solo ed esclusivamente sull'esecuzione dell'applicazione. Ciò comporta che, non dovendo avere a che fare con la gestione dei server su cui viene messa in campo l'applicazione, lo sviluppatore possa focalizzarsi sull'innovazione dell'applicazione e la messa in campo del prodotto venga accelerata

Il *function-as-a-service* si basa invece sul concetto di *funzione*, che viene eseguita alla ricezione di uno o più eventi e che ne gestirà l'elaborazione e un eventuale risposta. Gli sviluppatori con questo tipo di approccio permettono di concentrarsi solo sulla scrittura e configurazione della funzione, in quanto la messa in campo verrà poi demandata al provider che avrà il compito di allocare le eventuali risorse necessarie per la sua esecuzione. Questo tipo di modello permette quindi una riduzione dei costi sulla manutenzione o acquisto dell'infrastruttura, poichè si andrà a pagare proporzionalmente all'effettivo utilizzo delle risorse. Tuttavia questo tipo di approccio presenta alcuni svantaggi. In primis, avendo a che fare con l'utilizzo di *funzioni* porta alla necessità di dover utilizzare un meccanismo di storage persistente per supportare il flusso dell'applicazione e collegare gli eventi tra di

loro. Inoltre, a causa dell'allocazione e de-allocazione delle risorse, si ha un una possibile perdita di performance che causa un ritardo sull'avvio della funzione, chiamata *cold start* il quale aggiunge di conseguenza una latenza.

## 1.2 Obiettivo Tesi

L'obiettivo finale di questo progetto è idealmente quello di ottimizzare il *Total Cost of Ownership* grazie all'ecosistema *FaaS*. Si utilizzerà Amazon Web Service come cloud provider, con il fine di confrontarne i servizi disponibili (AWS Lambda, Aurora Serverless) con Knative, una soluzione open-source che permette l'uso dell' ambiente sopra citato. A tale scopo, si studieranno dapprima i servizi messi a disposizione da AWS e il framework Knative, per poi implementare un' applicazione a microservizi, cosı da permettere un confronto a livello sia di prestazioni sia di costo di queste tecnologie, volto allo scoprire se Knative possa essere una valida alternativa per l'ecosistema FaaS. La tesi verrà quindi organizzata nei seguenti capitoli:

- **Capitolo 2**: Si analizzerà lo stato dell'arte, il cui focus sarà l'analisi delle alternative a *Knative* elencando i benefici e i contro di queste tecnologie.
- **Capitolo 3**: Si andranno a presentare *Docker* e *Kubernetes* indicando i loro componenti e il loro funzionamento.
- **Capitolo 4**: Si presenterà *Knative* indicando la sua architettura e il suo funzionamento all'interno di Kubernetes.
- **Capitolo 5**: Si descriverà *Amazon Web Services* e i suoi servizi.
- **Capitolo 6**: Si descriverà l'applicazione web creata.
- **Capitolo 7**: Si forniranno i test effettuati sull'applicazione.
- **Capitolo 8**: Si andranno ad analizzare i test effettuati e verranno fatti dei calcoli con lo scopo di vedere se c'è un effettivo risparmio di costi con le diverse tecnologie applicate.

# Capitolo 2

# Stato dell'arte

In questo breve capitolo verranno trattati, dal punto di vista teorico, quali sono i maggiori competitor con cui si deve confrontare *Knative*.

## 2.1 Kubeless

*Kubeless* [3] è un framework open-source basato su *Kubernetes*. Consente di sfruttare le sue risorse per fornire la scalabilità automatica, monitoraggio, API routing e troubleshooting. L'entità base di Kubeless sono le **function**, le quali permettono di mettere in campo piccole porzioni di codice senza doversi occupare dell'infrastruttura sottostante. Le *function* possono essere eseguite attraverso la CLI fornita durante l'installazione di Kubeless, oppure, nel caso di *function* più complesse, attraverso file **YAML**. La parte principale di un framework serverless riguarda l'autoscaling. Kubeless si appoggia all'autoscaling fornito da Kubernetes chiamato **HorizontalPodAutoscaler**. In Kubeless ogni funzione viene eseguita come un deployment di Kubernetes separato, il che lascia una totale libertà all' *HPA* di scalare automaticamente la funzione in base alla metrica che fornisce durante la sua esecuzione. La differenza principale rispetto a *Knative* è che quest'ultimo non supporta lo scaling automatico a zero e non consente di sviluppare applicazioni basate su container in modalità serverless.
Un altro aspetto molto importante riguarda il suo utilizzo all'interno dei reparti IT. Ad oggi *Kubeless* non è più mantenuto e aggiornato dai creatori (BitMap). Il continuo rilascio di aggiornamenti e di una community sempre attiva è il fulcro su cui si basano le applicazioni open-source ed è sopratutto per questo motivo che le aziende IT stanno incominciando a muovere i primi passi verso *Knative* grazie al costante supporto e ai suoi continui aggiornamenti forniti da Google.

### 2.1.1 Funzioni

Come descritto precedentemente, le funzioni denominate *Function* in *Kubeless* rappresentano l'entità principale di questo framework.Per default, le function vengono eseguite come utente non privilegiato, in questo modo non si dovranno inserire dei permessi di root. L'esecuzione può comunque essere modificato interagendo con il *Security Context* all'interno del file di configurazione della funzione. I parametri principali della *function* sono:

- **Timeout**: Si specifica il timeout massimo della funzione, al termine l'esecuzione della funzione termina.
- **Runtime**: Si indica che runtime dovrà utilizzare specificandone la sua versione. Le runtime dovranno essere quelle offerte da Kubeless.
- **Handler**: Consente di specificare che funzione si andrà a chiamare all'interno di un file. La notazione è *nomefile.nome_Funzione*.
- **Deps**: Indica quali dipendenze avrà la funzione.
- **Function Content Type**: Indica il tipo della funzione. I formati supportati sono: base64,url e text.
- **Function**: Indica il contenuto della funzione: Codice o URL.

## 2.2 OpenFaas

*OpenFaas* è un framework open-source serverless, la sua caratteristica principale consiste nella sua facilità di utilizzo e della sua portabilità in quanto è una delle poche piattaforme capaci di funzionare su orchestratori diversi da *Kubernets* come ad esempio *Docker Swarm*.

### 2.2.1 Architettura

Rispetto a *Knative*, nel quale la sua natura risiede sull'essere un componente on top di *Kubernetes*,*OpenFaas* dispone di una sua architettura che gli permette di poter comunicare con diversi orchestratori come indicato nella figura [4].

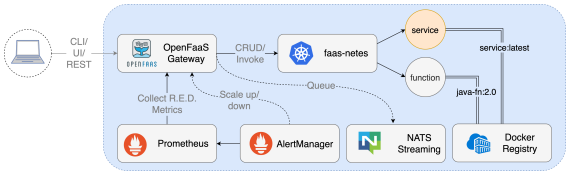



Figura 2.1: Architettura di OpenFaaS

Il primo componente che si incontra quando si vuole accedere al framework è il *Gateway*. Questo componete è reso accessibile attraverso delle chiamate API REST, CLI o interfaccia web prioritaria fornita durante l'installazione.Il **Gateway** fornisce una serie di API che permettono la comunicazione con il componente *faas-netes*, un provider di *OpenFaaS* per *Kubernetes*. Questo componente una volta ricevuta la richiesta da parte del gateway avrà il compito di eseguire la funzione richiesta appoggiandosi ad un *Docker Registry* (vedere sezione 3.1.2). La separazione tra il gateway e *faas-netes* permette a OpenFaaS di non essere dipendente da un singolo orchestratore in quanto basterà cambiare il componente *faas-netes* con un'altra implementazione al fine di poter comunicare con un altro orchestratore diverso da *Kubernetes*. Per quanto riguarda lo scaling, non viene utilizzato quello di Kubernetes ma viene gestito nativamente da OpenFaaS.In particolare, viene utilizzato *prometheus*, un software utilizzato per il monitoring che ha il compito di collezionare le metriche fornite dal framework e di notificare il gateway di OpenFaaS al fine di far scalare in alto o in basso le repliche disponibili nel cluster. *Prometheus* prima di notificare il gateway controlla il componente chiamato **AlertManager**, un insieme di regole definite dall'utente, che se violate consente la comunicazione con il gateway e far partire lo scaling di OpenFaaS. Come succede in *Knative*, l'autoscaling consente di andare anche a zero, nel caso di OpenFaaS questa funzione potrà essere abilitata solo con **OpenFaaS Pro** attraverso il componente denominato **faas-idler**.

## Funzioni

Come avviene con *Kubeless*, il framework lavora attraverso delle funzioni e non attraverso i container come viene gestito in *Knative*. In questo framework, la creazione di una *function* viene realizzata sfruttando dei template disponibili attraverso la CLI fornita da OpenFaaS oppure attraverso una dashboard integrata nell'interfaccia web. I template permettono di avere diversi ambienti nei quali è possibile scrivere codice nei linguaggi più popolari. Ogni funzione, oltre al processo responsabile della sua esecuzione è affiancato da un altro processo chiamato **watchdog**.

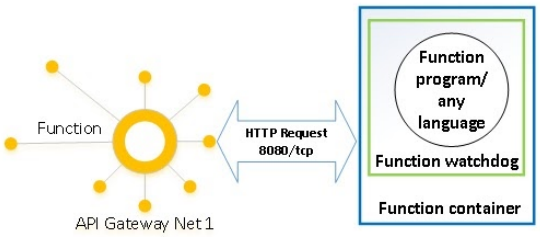



Figura 2.2: Interfaccia OpenFaaS Wathdog

# Capitolo 3

# Strumenti utilizzati

Nel capitolo a seguire concentreremo la nostra analisi su due strumenti cardine per questa tesi. **Docker** e **Kubernetes**.

## 3.1 Docker

*Docker* [5] è strumento sviluppato nel 2013 come un progetto interno open-source da parte di *dotCloud*, un'azienda focalizzata sul modello *Platform-as-a-Service* (vedere sezione 1.1). La tecnologia di *Docker* consente l'esecuzione di applicazioni rendendone più semplice il loro processo di sviluppo e la loro distribuzione. Per farlo, impacchetta queste applicazioni in delle unità standardizzate chiamate **container** che, come vedremo successivamente (sezione 3.1.1) , offrono tutto il necessario per la corretta esecuzione dell'applicazione includendo all'interno di questi: librerie,strumenti di sistema e codice. Al momento, risulta essere una delle tecnologie open-source più utilizzate per creare,testare e distribuire container con la massima rapidità e semplicità da parte degli sviluppatori. In aggiunta, i container possono essere eseguiti ovunque su qualsiasi macchina.Questa caratteristica di *Docker* permette di distinguersi dai *Linux Container* in quanto i container dovranno ogni volta essere adattati per la macchina su cui andranno ad essere eseguiti. Di seguito, alcuni aspetti fondamentali di *Docker*:

- **Leggero**: *Docker* lavorando sulla virtualizzazione al livello del sistema operativo rende possibile l'esecuzione di più container sulla stessa macchina condividendone lo stesso kernel. Per permettere un tempo di avvio veloce dei container,invece di scaricare l'intera immagine, attraverso una visione stratificata del file system chiamata *Union File System* è possibile condividere tra i vari container disponibili i *base-file system* permettendo di poter scaricare solo una porzione di quella immagine e avviandolo molto più velocemente.

- **Portatile**: Come descritto in precedenza, la tecnologia *Docker* permette di creare e distribuire applicazioni basata a container da eseguire su qualunque dispositivo.

- **Isolato**: Come avviene anche per i LXC, i *Docker container* sono isolati tra di loro durante la loro esecuzione grazie alla possibilità di utilizzare i namespace e i cgroups forniti dal kernel Linux.La loro separazione permette che in caso di problemi riguardante uno o più container non vada ad intaccare quelli che funzionano in modo corretto.

### 3.1.1 Docker Container

Un *container* contiene tutto quello di cui l'applicazione ha bisogno per essere eseguita permettendolo di farlo in modo isolato da tutti gli altri container all'interno della macchina.Poichè sfrutta il kernel Linux, un container Docker a differenza di una macchina virtuale non contiene alcun sistema operativo al suo interno.

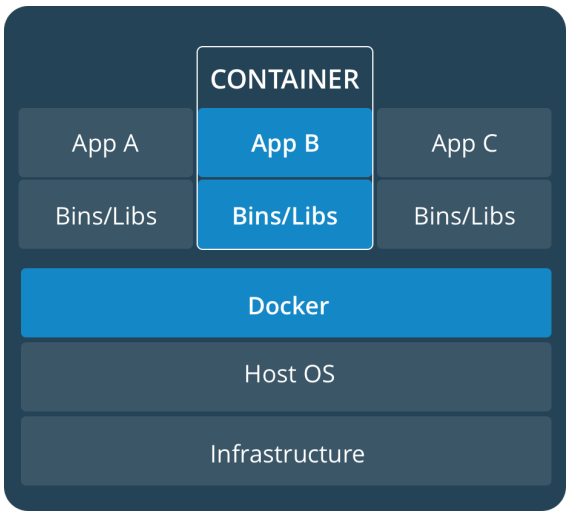



Figura 3.1: Container Docker

La tecnologia *Docker* è stata sviluppando partendo dalla tecnologia LXC. Questa tecnologia di partenza consente una virtualizzazione leggera ma senza offrire un'esperienza ottimale agli sviluppatori e agli utenti [6].I container Linux adottano un sistema **init** in grado di gestire più processi con lo scopo di eseguire più applicazioni come una singola entità. *Docker* invece, incoraggia a suddividere le applicazioni in microservizi fornendo gli strumenti per poterlo fare garantendo numerosi vantaggi.

### 3.1.2 Architettura

*Docker* è composto da diversi componenti:**Docker Client**,**Docker Images**,**Docker Registry**,**Docker Engine**.

#### Docker Engine

*Docker Engine* costituisce il *core* della piattaforma comportandosi come un'applicazione client-server la quale espone una serie di chiamate API di tipo REST che i vari programmi potranno sfruttare per la comunicazione con il *Docker daemon* che ha il compito di mettere in campo,eseguire e distribuire i container *Docker* [7].

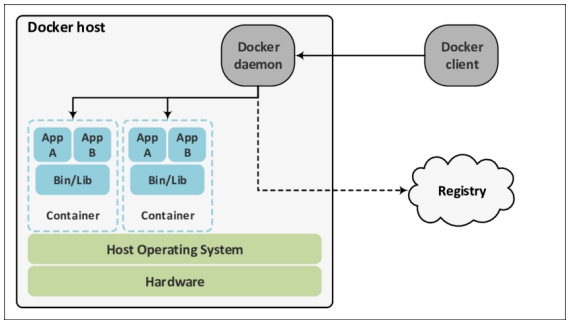



Figura 3.2: Architettura Docker

#### Docker Images

*Docker* fornisce un modello basato a immagini per poter distribuire applicazioni o un insieme di servizi.Ogni immagine ha la caratteristica di essere solo un template *read-only*, al suo interno vengono inseriti un'insieme di file e parametri con lo scopo di definire e configurare l'applicazione. Successivamente, attraverso l'immagine sarà possibile la creazione di un container.

#### Docker Client

Facendo riferimento alla figura 3.2 si tratta di un componente che ha il compito di interfacciarsi con le API fornite dal *Docker Engine* attraverso una serie di comandi disponibili nella *Command Line Interface*. Ogni comando inizia con la parola chiave *docker* e permette ad esempio di poter creare una immagine attraverso un *DockerFile* o far partire l'esecuzione di un container nel caso in cui sia già presente un'immagine.

### Docker Registry

L'immagine che *Docker* crea sono inserite successivamente in una registro chiamato **Docker Registry**.Il suo funzionamento è simile ad un repository di un codice sorgente ( come ad esempio *github* ) dove, al posto del codice, verranno inserite le immagine create precedentemente. Ci sono due tipologie di registro: pubbliche e private. Uno dei principali registri pubblici è *Docker Hub* un sito web nel quale tutti gli utenti possono estrarre le immagini disponibili o inserire le proprie rendendole disponibili a tutti. L'immagine viene caricata all'interno di un registro attraverso il comando *docker push* mentre viene scaricata attraverso il comando *docker pull* sempre tramite la CLI.

## 3.1.3 Dockerfile

L'immagine per essere costruita da *Docker* attraverso il comando *docker build* ha bisogno di un file nel quale viene descritto il contenuto e la relativa configurazione dell'applicazione. Il file in questione viene chiamato **Dockerfile**, al suo interno vengono inserite un'insieme di istruzioni tali per per cui poco a poco permetta la corretta costruzione dell'immagine. Le istruzioni principali sono:

- **FROM**: É un'istruzione obbligatoria in ogni *Dockerfile* e dovrà anche essere la prima istruzione presente all'interno del file. Questo comando va a definire l'immagine base su cui si andrà a definire la propria.
- **COPY**: Permette di poter copiare dei file locali all'interno della propria immagine in una posizione che dovrà essere specificata dallo sviluppatore.
- **ENV**: Permette di poter definire delle variabili d'ambiente che verranno utilizzate all'interno del container.
- **RUN**: Esegue nel *Docker Engine* un comando shell con lo scopo di installare software e package con il fine di eseguire l'applicazione desiderata.
- **CMD**: Permette di poter eseguire dei comandi di shell subito dopo che il container viene avviato.
- **EXPOSE**: Permette di dichiarare quali porte sono raggiungibili dall'esterno.

*Docker* fornisce uno strumento utile per poter unire quelle applicazioni o servizi che potranno essere eseguiti insieme in un ambiente isolato. *Docker Compose* permette, attraverso la stesura di un file YAML, la possibilità di configurare ed eseguire i servizi dell'intera applicazione attraverso un singolo comando. Il file, chiamato *docker-compose.yaml*, ha una struttura chiara nel quale si definiscono una lista di servizi che verranno rilasciati come dei *docker container*.

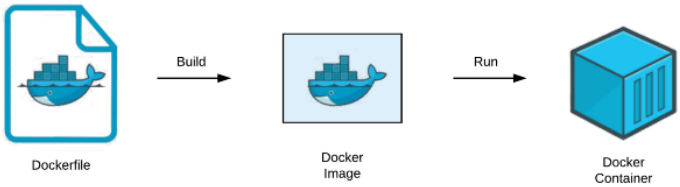



Figura 3.3: Funzionamento Dockerfile

### 3.1.4 Docker network

Ogni *Docker container* ha una sua network interface nella quale viene assegnato un indirizzo IP dinamico.L'indirizzo associato ad ogni *Docker container* non ha nulla a che vedere con la rete fisica su cui viene eseguito. Il motivo è dovuto all'indirizzo IP, in quanto non viene fornito dal DHCP in modo automatico ma da un componente di Docker che consiste in una libreria chiamata **Libnetwork**. La ragione di questa scelta è dovuta ad una questione di leggerezza e modificabilità. Rispetto ad un server DHCP classico, il componente viene creato utilizzando il *Container Network Model* che permette a *Docker* di fornire una specifica interfaccia per poterlo sostituire con uno creato dallo sviluppatore permettendo di avere il pieno controllo della rete nel caso in cui ce ne fosse il bisogno. I container sono collegati tra di loro attraverso un componente definito da Docker chiamtato **docker0**.I vari container, i quali dispongono di un indirizzo IP privato, utilizzano un software chiamato **iptables** che permette la comunicazione con il mondo esterno utilizzando un insieme di regole NAT configurato da Docker stesso .

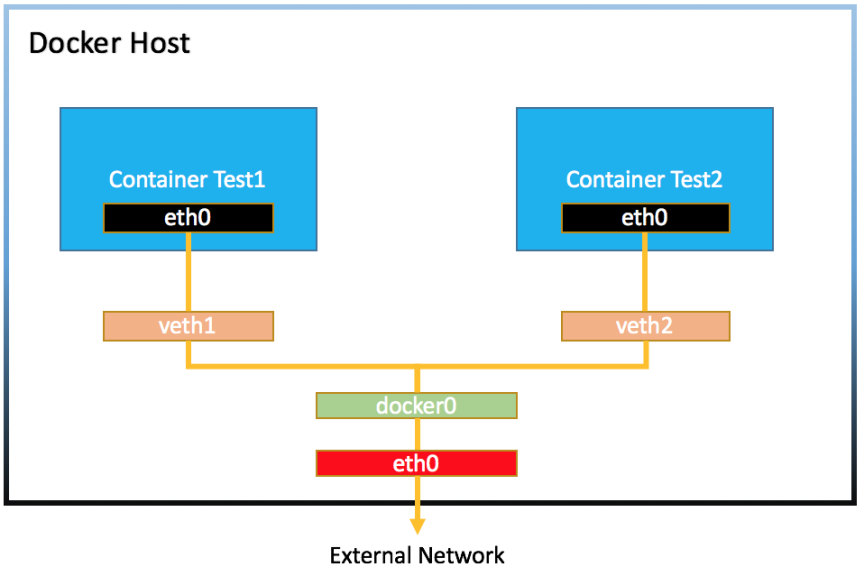



Figura 3.4: Docker Network

## 3.2 Kubernetes

*Kubernetes* è una piattaforma portatile,estensibile e open-source per la gestione di carichi di lavoro e servizi a container, in grado di facilitarne sia la configurazione dichiarativa che l'automazione[8].La sua origine proviene da un sistema interno sviluppato da Google chiamato Borg il quale permetteva di aiutare gli sviluppatori e gli amministratori di sistema a gestire numerose applicazioni e servizi. I container,come abbiamo potuto notare nella sezione relativa a Docker, sono un ottimo strumento per poter distribuire ed eseguire le proprie applicazioni.In un ambiente di produzione i container non possono essere gestiti solamente da Docker in quanto, in caso di interruzione di un servizio, dovrà essere gestito in modo manuale da parte di un utente.D'altro canto, *Kubernetes* ci viene in soccorso in quanto fornisce un framework per far funzionare i sistemi distribuiti in modo resiliente e automatico. In particolare:

- **Scoperta dei servizi e bilanciamento di carico**: Un container può essere esposto utilizzando un DNS oppure il suo indirizzo IP. Nel caso in cui il traffico verso il container è alto, Kubernetes è in grado di distribuire il traffico su più container in modo tale che il servizio rimanga stabile.
- **Orchestrazione dello storage**: Kubernetes offre la possibilità di montare in modo automatico un sistema di storage scelto dall'utente.
- **Rollout e rollback automatizzati**: Durante la messa in campo del container è possibile inserire lo stato desiderato, Kubernetes,in modo automatico, si occuperà di raggiungerlo.
- **Ottimizzazione dei carichi**: Ad ogni container è possibile inserire quante risorse di CPU e RAM serviranno per la sua corretta esecuzione.Kubernetes si occuperà di allocare i container sui nodi in modo tale da poter ottimizzare l'uso delle risorse.
- **Self-healing**: Nel caso in cui un container si blocca o non risponde più, Kubernetes si occuperà di riavviarlo,sostituirlo o terminarlo.
- **Gestione dei segreti e della configurazione**: All'interno di Kubernetes è possibile memorizzare e gestire delle informazioni sensibili come password o chiavi SSH.

### 3.2.1 Architettura

Un cluster *Kubernetes* è un'insieme di macchine,chiamate nodi, che eseguono container gestiti da Kubernetes. Ogni cluster ha almeno un nodo chiamato **Worker Node** nel quale al suo interno viene ospitato il **control plane**. All'interno del/dei worker node vengono

inseriti i Pod che avranno il compito di eseguire i carichi di lavoro dell'utente. In un ambiente di produzione, il control plane viene eseguito su più macchine in modo tale da fornire un'alta tolleranza ai guasti.

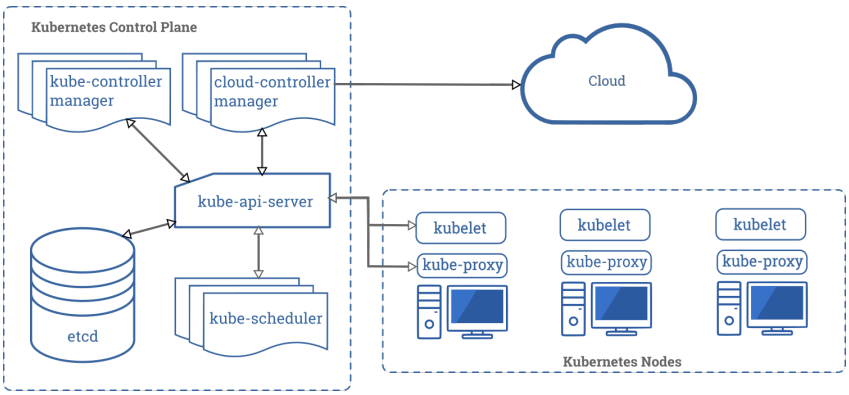



Figura 3.5: Componenti di Kubernetes

### 3.2.2 Componenti control plane

Il *control plane* vanta di diversi componenti al suo interno. Il loro scopo è quello di gestire in modo automatico i container, in particolare, si occupa di rilevare e rispondere agli eventi all'interno del cluster.

#### kube-apiserver

La parte principale del control plane di Kubernetes è API server.Espone un'API HTTP consentendo agli utenti,alle diverse parti del cluster e ai componenti esterni di poter comunicare tra di loro. Le API consentono di interrogare e manipolare lo stato degli oggetti di Kubernetes come POD,namespace, ConfigMaps e Events.La maggior parte delle operazioni viene eseguita tramite l'interfaccia a riga di comando *Kubectl* tuttavia, è possibile accedere direttamente all'API utilizzando delle chiamate REST. Nel control plane, il componente incaricato a fornire queste API è chiamato **kube-apiserver** ed è progettato per scalare orizzontalmente aumentando il numero di istanze con il fine di bilanciarne il traffico.

#### etcd

É un database key-value, usato da Kubernetes per salvare tutte le informazioni del cluster. Questo database può essere installato all'interno del *master node* oppure,attraverso

un dispositivo esterno.In ogni caso, solo attraverso le API fornite da *kube-apiserver* sarà possibile avviare una comunicazione con il database.

#### Scheduler

Ha il compito di assegnare i processi detti Pod a dei nodi. I fattori su cui si basa la scelta del nodo è in base alle risorse che il pod chiede e ai vincoli hardware dei vari nodi disponibili. Il componente fornito da Kubernetes è chiamato **kube-scheduler** ma, è possibile inserirne un altro indicando ai pod di utilizzarlo.

#### kube-controller-manager

É un componente che ha il compito di comportarsi come un controllore. Attraverso l'utilizzo di API è in grado di poter osservare lo stato del cluster e nel caso in cui ce ne fosse il bisogno di effettuare delle modifiche per effettuare la transizione dallo stato attuale allo stato desiderato delle varie applicazioni. Alcuni controller gestiti da questo componente sono:

- **Node Controller**: Monitora il nodo del cluster.
- **Replication Controller**: Mantiene il numero corretto di Pod per ogni *ReplicaSet* presente nel sistema.
- **Endpoints Controller**: Ha il compito di collegare i pod con i services.
- **Service Account & Token Controllers**: Creano account di defautl e i token di accesso alle API per i nuovi namespaces.

#### cloud-controller-manager

Si tratta di un componente che ha lo scopo di aggiungere delle logiche di controllo specifiche per il cloud. In particolare, permette di collegare il proprio cluster con le API fornite dal proprio cloud provider separando i componenti che interagiscono con il cloud con quelli che interagiscono solamente con il cluster. Nel caso in cui Kubernetes non viene installato su un cloud provider il componente non viene installato. Come avviene nel kube-controller-manager, il componente combina diversi control loop logicamente indipendenti che però vengono eseguiti come un singolo processo. Questa soluzione permette lo scaling orizzontale per migliorare la responsività e la tolleranza ai guasti. I controller che comunicano attraverso questo componente sono:

- **Node Controller**: Controlla i nodi nel cloud provider se hanno smesso di funzionare.

- **Route Controller**: Configura le network route nell'infrastruttura cloud.
- **Service Controller**: Crea, aggiorna ed elimina i load balancer forniti dal cloud provider.

### 3.2.3 Componenti dei Nodi

I componenti che andremo ad elencare in questa sezione vengono eseguiti in ogni nodo in modo tale da poter mantenere i processi in esecuzione.

#### Kubelet

Tiene sotto controllo lo status dei processi avviati nel nodo. Questo componente, riceve un set di *PodSpecs* i quali vengono forniti attraverso vari meccanismi. Kubelet si assicura che i container descritti in questi *PodSpecs* funzionino in modo corretto, nel caso in cui ci siano dei container che non sono stati creati da Kubernetes, kubelet non li gestirà e non garantirà la loro corretta esecuzione.

#### Kube-proxy

É un proxy eseguito su ogni nodo del cluster, mantiene le regole di networking su nodi permettendo la comunicazione tra di loro e all'esterno. Utilizza le librerie del sistema operativo quando possibile altrimenti questo componente gestirà il traffico direttamente.

#### Container Runtime

Software responsabile per l'esecuzione dei container.Kubernetes al momento supporta diversi container engine come *Docker*, *containerd*,*cri-o*,*rklet* e tutte le implementanzioni di *Kubernetes CRI*.

#### Addons

Questi componenti utilizzano risorse Kubernetes quali *DaemonSet,Deployment* per implementare funzionalità di cluster.Queste funzionalità non sono native di Kubernetes ma vengono implementate da aziende third-party attraverso dei pod. I pod vengono inseriti all'interno del namespace *kube-system*. Alcuni addons più famosi sono *DNS* che consiste in un server DNS aggiuntivo e *Dashboard*, un'interfaccia web che permette agli utenti di gestire e fare trubleshooting delle applicazioni che girano all'interno del cluster.

### 3.2.4 Oggetti di Kubernetes

Kubernetes utilizza queste entità per rappresentare lo stato del cluster, in particolare descrivono:

- Quale applicazione viene eseguita e in quale nodo.
- Quali risorse sono disponibili per quelle applicazioni.
- Le policy a cui le applicazioni sottostanno.

Come detto nella sezione 3.2.2 , per la creazione, modifica e l'eliminazione degli oggetti, kubernetetes dispone delle API attraverso l'interfaccia a riga di comando Kubectl il quale attraverso un file in formato YAML sarà possibile esprimere lo stato desiderato dell'oggetto. Gli oggetti Kubernetes includono due campi che regolano la configurazione dell'oggetto in questione [9]:

- **Spec**: Viene impostato alla creazione dell'oggetto, fornendo una descrizione delle caratteristiche che si vogliono ovvero lo stato desiderato.
- **Status**: Descrive lo stato corrente dell'oggetto, viene fornito e aggiornato da Kubernetes. Il control plane gestisce continuamente e attivamente lo stato effettivo di ogni oggetto in modo tale da essere nello stato desiderato descrito dal campo "spec".

Se si vuole creare un oggetto Kubernetes è importante che all'interno del file YAML vengano inseriti i seguenti campi:

- **apiVersion**: Indica la versione di API con cui comunicherà con Kubernetes.
- **kind**: Una stringa che rappresenta il tipo di oggetto che si andrà a creare.
- **metadata**: Informazioni come nome, UID e namespace per identificare l'oggetto all'interno del cluster.
- **spec**: Identifica lo stato desiderato per l'oggetto.

#### Namespace

Può essere visto come un cluster virtuale. Il loro utilizzo permette di distinguere diversi insieme tra i pod nel caso in cui l'ambiente di Kubernetes è condiviso tra molti utenti. I namespace non possono essere annidati l'uno dentro l'altro. All'interno del namespace viene costruito un ambiente nel quale i nomi delle risorse dovranno essere univoci in modo da permettere di dividere logicamente il cluster di Kubernetes in gruppi di risorse. Con l'installazione di Kubernetes vengono definiti diversi namespace:

- **kube-system**: Contiene gli oggetti creati da Kubernetes, in particolare quelli del control plane.

- **default**: Contiene gli oggetti e le risorse che vengono create dall'utente.

- **kube-public**: É leggibile da tutti gli utenti ( anche quelli non autenticati) ed è riservato per contenere degli oggetti che servono per tutto il cluster.

- **kube-node-lease**: Contiene al suo interno oggetti *Lease* associati ad ogni nodo. Il loro compito è quello di mandare degli stati di salute riguardo il nodo al componente. *kubelet* in modo tale che il control plane capisca quando avviene un guasto.

## Volume

I volumi risolvono il problema della persistenza dei dati tra container all'interno dello stesso Pod. I file all'interno dei container sono effimeri e per questo motivo nel caso in cui ci sia un'interruzione del container questi file andranno persi. Un volume di Kubernetes è un componente di un pod ed è quindi definito durante la creazione all'interno del file YAML [10].Non è un oggetto di Kubernetes autonomo e non possono essere creati e distrutti da soli in quanto condivide lo stesso ciclo di vita del Pod a cui è associato. Un volume è disponibile per tutti i container all'interno del pod ma dovrà essere montato in un container. Kubernetes offre anche un modo per essere indipendente dal ciclo di vita del Pod introducendo il concetto di Persistent Volume.

### Pod

Un pod è l'unità più piccola che si può creare all'interno di Kubernetes, rappresenta un processo all'interno del cluster. Incapsula al suo interno, uno o più containar i quali condividono diverse risorse quali: memorizzazione dei dati, rete e le specifiche per descrivere il funzionamento dell'applicazione. Non è solita la creazione di un Pod direttamente da Kubernetes in quanto di solito viene creato attraverso altri tipi di risorse come Deployment,StatefulSet o DaemonSet i quali gestiscono l'intero ciclo di vita del Pod. Il modello one-container-per-Pod è il più utilizzato in quanto permette di visualizzare il pod come un wrapper intorno al semplice container. Nel caso in cui si abbiano più container all'interno di un singolo pod ,molto spesso, cooperano tra di loro.In questo caso, i container vengono schedulati insieme sulla stessa macchina del cluster e potranno comunicare tra di loro utilizzando la connessione localhost e condividere i dati attraverso l'oggetto Volume, questi container che cooperano con quello principale sono chiamati **sidecars**.

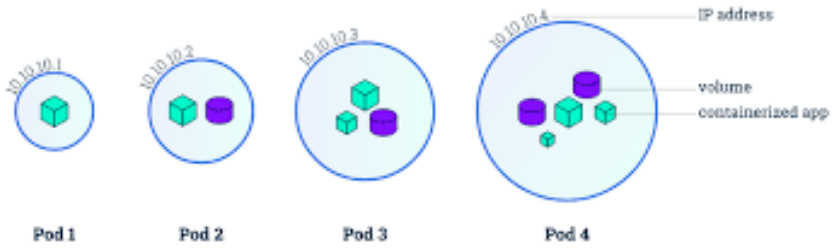



Figura 3.6: Kubernetes Pod [11]

### DaemonSet

É un pod speciale il quale serve per assicurare l'esecuzione di ogni pod in tutti i nodi del cluster o di un suo sottoinsieme. Ad ogni inserimento o rimozione di un nodo all'interno del cluster, il DaemonSet si occupa di aggiungere o rimuovere il pod specificato.

### StatefulSet

Si tratta di un pod con il compito di gestire le applicazioni in modalità stateful. Questa gestione avviene su un insieme di pod fornendo lo scaling e garantendo l'ordinamento e l'unicità. In modo molto simile ad un Deployment, uno StatefulSet gestisce dei pod basati sulle stesse specifiche di un container,ma con l'aggiunta che mantiene un'identità univoca e permanente per ciascuno dei suoi pod.

### ReplicaSet

In Kubernetes possiamo avere numerose repliche dei nostri èod. Esistono due tipi di approcci per gestire queste repliche, la prima è un approccio tradizionale detto "Pets model" nel quale i server sono unici e indispensabili. Con questo approccio se un server smette di funzionare si cercherà di eseguire dei fix per rimetterlo in piedi e farlo funzionare. Il secondo metodo invece chiamato "Cat model" ed è quello che utilizza il ReplicaSet, i server sono visti come repliche all'interno di un gruppo di server, se un server smette di funzionare verrà distrutto e ne verrà replicato uno nuovo identico. Questo concetto si basa sul fatto che i server sono designati per fallire prima o poi.Il ReplicaSet, controlla continuamente che le repliche che lui gestisce siano vive.

### Deployment

Un Deployment fornisce un metodo dichiarativo per la creazione e la gestione dei pod e dei ReplicaSet.All'interno del Deployment viene descritto lo stato desiderato, successivamente il Deployment controller si occuperà di modificare l'attuale stato per arrivare allo stato desiderato.

```
1  kind: Deployment
2  metadata:
3    name: nginx-deployment
4  spec:
5    selector:
6      matchLabels:
7        app: nginx
8    replicas: 2 # tells deployment to run 2 pods matching the
       template
9    template:
10     metadata:
11       labels:
12         app: nginx
13     spec:
14       containers:
15       - name: nginx
16         image: nginx:1.14.2
17         ports:
18         - containerPort: 80
```

Listato 3.1: Esempio Kubernetes Deployment

### Service

Un Service in Kubernetes rappresenta un meccanismo per fornire l'esposizione verso il mondo esterno di un'applicazione il quale viene eseguito in un'insieme di pod. Questa astrazione disaccoppia l'esposizione dei pod dalla loro esecuzione. Ogni Service ha un proprio indirizzo IP e una porta che non cambierà mai fino a quando quel determinato servizio non cesserà di esistere. In questo modo, il concetto di disaccoppiamento si farà più interessante in quanto i pod potranno essere aggiornati,eliminati o creati senza doversi preoccupare di cambiare indirizzo IP e porta. Kubernetes offre diversi tipi di Service:

- **ClusterIP**: Espone il Service attraverso un indirizzo IP interno al cluster, questo rende raggiungibile il servizio solo all'interno del cluster.

- **NodePort**: Espone il Service su ogni indirizzo IP dei nodi su una porta statica (NodePort) uguale per tutti i nodi. Con questa soluzione è possibile collegarsi al nodo dall'esterno inserendo l'indirizzo IP del nodo più la porta: IP:Porta.
- **LoadBalancer**: Espone il Service all'esterno utilizzando un bilanciatore di carico fornito da un cloud provider. Il loadbalancer può essere raggiunto attraverso un VirtualIP.

Esiste un altro modo per poter esporre un Service ed è attraverso l'utilizzo di un Ingress. Esso, si comporta come un punto di ingresso al cluster permettendo la scrittura di regole per il routing e l'esposizione di differenti servizi su uno stesso indirizzo IP.

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4   name: nginx-service
5   labels:
6     app: nginx
7  spec:
8   type: ClusterIP
9   selector:
10    app: nginx
11  ports:
12  - port: 80
13    targetPort: 80
```

Listato 3.2: Esempio Kubernetes Service

### Ingress

Una ragione importante sull'utilizzo di un Ingress controller è che ogni Service fornito da un Loadbalancer richiede un proprio bilanciamento di carico con il proprio indirizzo IP pubblico mentre un ingress ne richiede solo uno anche in caso di servizi divevrsi.Per sfruttare le risorse Ingress, è necessario che all'interno del cluster di Kubernetes sia presente un Ingress Controller. I controller non sono disponibili in modo automatico da Kubernetes durante l'avvio, ma sarà necessario installare un'implementazione diversa del controller a seconda dell'ambiente in cui si andrà a lavorare nel cluster. Attualmente Kubernetes supporta i controller GCE e nginx ma supporta anche altri controller come Ambassador,Gloo,Istio,Kourier.Kubernetes inoltre, offre la possibilità di installare diversi controller contemporaneamente all'interno del cluster.

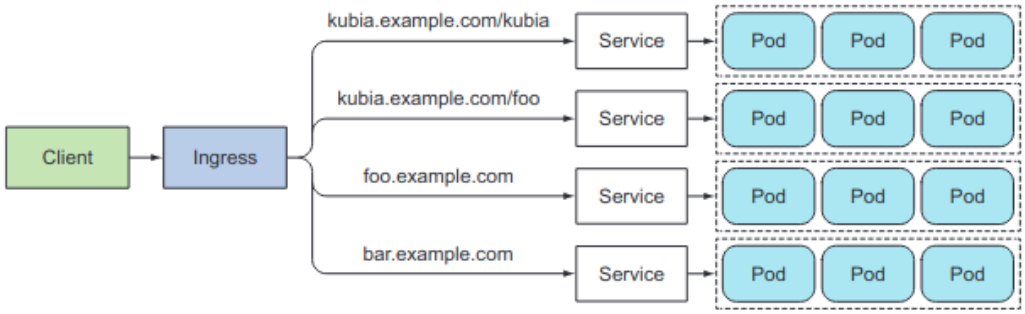



Figura 3.7: Kubernetes Ingress [10]

# Capitolo 4

# Knative

*Kubernetes* ,come si è potuto vedere nel capitolo precedente, risulta essere un ottimo strumento con un grande potenziale e,insieme a lui , anche l'ecosistema che lo circonda è in continua crescita e in espansione grazie ai molti strumenti e tecnologie in arrivo. In questa tesi, si è lavorato sullo studio e sull'utilizzo di **Knative**.
Knative è un framework serverless open source basato su Kubernetes , inizialmente sviluppato da *Google* per poi prenderne parte anche altri studi come *Red Hat,IBM,Pivotal e SAP*. La sua caratteristica principale consiste nel fornire un insieme di componenti con il fine di poter sviluppare applicazioni basate a container in modalità serverless che possano essere messe in campo in locale,in cloud oppure in datacenter di terze parti [12].
L'idea principale si basa sulla possibilità di permettere agli sviluppatori di focalizzare l'attenzione sulla scrittura del codice senza dover risolvere tutti i compiti più complessi per inserire e gestire l'applicazione.In particolare questi compiti possono essere riassunti in:

- Deploying di un container.
- Instradamento e gestione del traffico con un approccio blue/green.
- Scaling automatico e dimensionamento del carico di lavoro in base alla richiesta.
- Collegamento tra i servizi già avviati e i diversi sistemi ed eventi.

Al momento, *Knative* risulta essere il componente serverless più installato su Kubernetes con un percentuale del 27% mentre i suoi competitor si attestano al 10% per *OpenFaaS* e 5% per *Kubeless*[13]. La sua popolarità e il suo continuo supporto cresce sempre di più anche grazie alle caratteristiche che offre:

- Supporta servizi scalabili, sicuri e stateless in pochi secondi.
- Fornisce API con una elevata astrazione in grado di potersi interfacciare con le applicazioni più comuni.

- Portabilità, Knative viene installato come un componente on top di un'istallazione di Kubernetes. Questa possibilità permette di essere eseguito da qualsiasi parte evitando possibili blocchi da parte dei fornitori dei servizi.
- Esperienza di sviluppo senza interruzioni, supporta GitOps, DockerOps, ManualOps, ecc.
- Supporta molti strumenti e framework comuni come Django, Ruby on Rails, Spring e molti altri.


Figura 4.1: Architettura di Knative

Come è possibile vedere nella figura 4.1,Knative, al contrario di altri framework, necessita di un *Ingress* o *API Gateway* per lo smistamento delle richieste verso i servizi creati attraverso Knative. Durante l'installazione è possibile scegliere diversi network layer compatibili con Knative:

- **Istio**[14]:Istio è una service mesh raccomandata da Knative, la quale offre funzionalità di load balancing, autenticazione service-to-service, monitoring, e management del traffico.
- **Ambassador**[15]:É un Gateway API open-source basato su *Envoy*, permette di gestire solamente il traffico tra il cluster e l'esterno ed è quindi consigliato se non si vuole fare l'uso di una service mesh.
- **Contour**[16]: É un Ingress Controller open–source basato su Envoy il quale soddisfa tutte le specifiche di rete supportando tutte le feature offerte da Knative.

- **Kourier**[17]: É un Ingress controller basato su Envoy, sviluppato per poter funzionare con Knative. É un'alternativa leggera all'ingress controller di Istio.

In questo capitolo si andranno ad analizzare i due componenti principali che compongono Knative:

- **Knative Serving**: Esegue i container in modalità serverless consentendo la scalabilità automatica dei container tramite un modello basato sulla richiesta per mettere a disposizione i carichi di lavoro on demand.
- **Knative Eventing**: Fornisce delle primitive per la creazione di applicazioni event-driven, garantendo il disaccoppiamento delle sorgenti e dei consumatori di eventi.

## 4.1 Knative Serving

*Knative Serving* è un componente costruito on top di Kubernetes progettato per la messa in campo di container in modalità serverless. In particolare questo componente si occupa di:

- Deployment rapido di container serverless.
- Autoscaling, incluso la possibilità di poter scalare a zero i propri servizi.
- Supporto di molteplici network layer come: *Ambassador,Contour,Kourier,Gloo,Istio.*
- Snapshot del codice con le relative configurazioni.

### 4.1.1 Risorse

*Knative Serving* definisce una serie di oggetti come *Custom Resource Definition* (CRDs). Questi oggetti vengono utilizzati con lo scopo di definire e controllare il comportamento dei propri serverless workload all'interno del proprio cluster.

- **Service**:Questa risorsa chiamata SERVICE.SERVING.KNATIVE.DEV gestisce automaticamente l'intero ciclo di vita del carico di lavoro ( che può essere di un'intera applicazione ad una semplice funzione). Si occupa di controllare la creazione ed il ciclo di vita degli altri oggetti con il fine di garantire che la propria applicazione possegga una *Route*, una *Configuration* e una *Revision*.
- **Route**:La risorsa chiamata ROUTE.SERVING.KNATIVE.DEV fornisce un endpoint e un meccanismo per instradare il traffico verso le Revision disponibili. La configurazione di default permette che il traffico venga indirizzato alla versione più recente. In scenari più complessi,come ad esempio durante la fase di test, l'API supporta lo split del traffico su base percentuale tra le varie Revision disponibili per quell'applicazione.

- **Configuration**:La risorsa chiamata CONFIGURATION.SERVING.KNATIVE.DEV descrive l'ultimo stato desiderato del deployment cercando di mantenerlo durante tutta la sua esecuzione. Inoltre, crea e traccia le *Revision* man mano che lo stato desiderato viene aggiornato.
- **Revision**:La risorsa REVISION.SERVING.KNATIVE.DEV fornisce uno snapshot del codice e della sua configurazione. Viene creata ad ogni update della risorsa Configuration. Le Revision che non sono più accessibili attraverso la risorsa Route vengono eliminate insieme a tutte le risorse kubernetes sottostanti a essa.

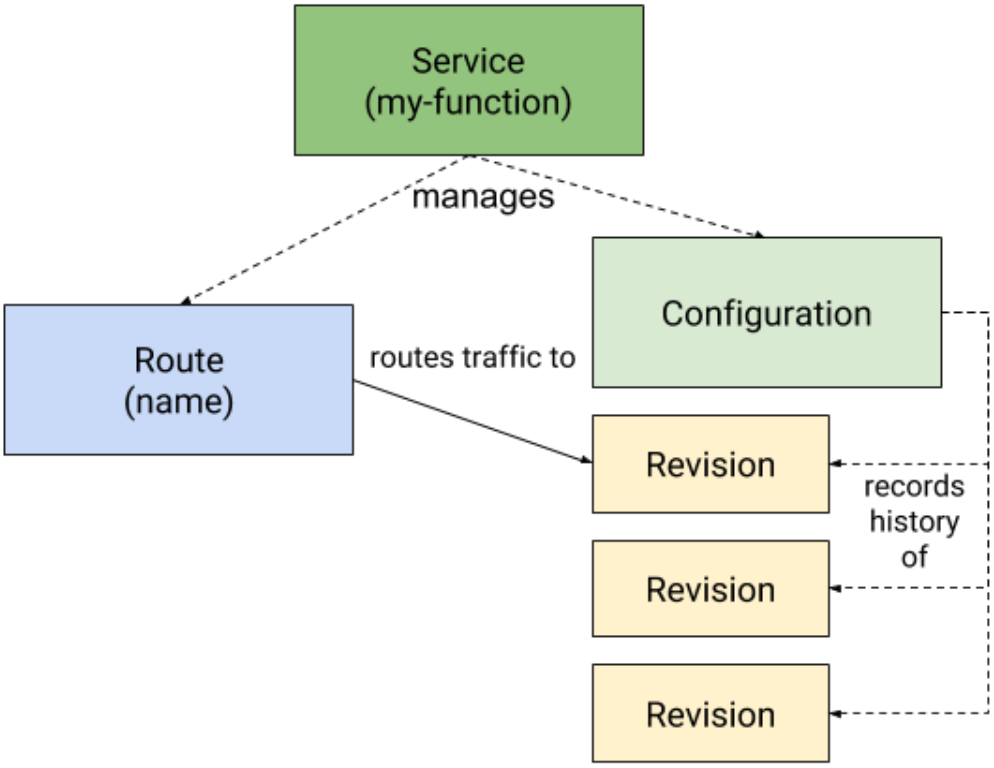



Figura 4.2: Struttura di Knative Serving

L'utilizzo di Knative Serving per il deployment parte da un'immagine *Docker* della propria applicazione, la risorsa che si andrà a creare risulterà più complessa di un *Service* in Kubernetes in quanto Knative si dovrà occupare di:

- Creare una *Revision* ad ogni nuova versione dell'applicazione.
- Creazione di risorse di rete quali *Route*,*Ingress* e l'alternativa del *Service* di Kubernetes.

- Scaling automatico del pod in relazione al traffico fino ad un numero massimo di repliche ed ad un minimo che può essere anche zero in assenza di traffico.

Nel momento in cui viene eseguito il deployment la risorsa richiederà un URL che verrà utilizzato per essere chiamato nel cluster.Per fare in modo che ogni Knative service disponga di un dominio , durante l'installazione del framework si dovrà andare a configurare un DNS come sslip.io che attraverso l'esecuzione del job *default-domain* permetterà di assegnare un URL temporaneo in modo automatico.Ovviamente, è consigliato l'utilizzo di un DNS vero e proprio nel momento in cui l'applicazione esce dalla fase di sviluppo.

```
1  kind: Service
2  metadata:
3   name: helloworld-go
4   namespace: default
5  spec:
6   template:
7    spec:
8     containers:
9      - image: gcr.io/knative-samples/helloworld-go
10       env:
11         - name: TARGET
12           value: "Go Sample v1"
```

Listato 4.1: Esempio Knative Serving

Per caricare una nuova applicazione, o più in generale un microservizio serverless, è necessario scrivere un file di configurazione YAML che verrà gestito direttamente dall'orchestratore, a differenza di Kubernetes, per il deployment di un *Knative Serving* si utilizzeranno le API relative alle Custom Resource introdotte da Knative. In particolare, nel campo relativo alle API da utilizzare per il deployment del servizio, è necessario specificare le API del componente Serving. Una volta scritto il file di configurazione dell'applicazione come mostrato sopra si dovrà eseguire il deployment attraverso il comando KUBECTL APPLY -F <FILE-YAML>.YAML. Il servizio verrà aggiunto alla lista dei servizi gestiti da *Knative* i quali attraverso il comando KUBECTL GET KSVC verranno forniti con l'URL e l'ultima revision disponibile. Successivamente, il comando CURL <URL> knative andrà a creare il pod il quale fornirà come output il messaggio "*Go Sample v1*". In base alle impostazioni dell'autoscaler (vedere sezione 4.1.3 ) il pod deciderà se cessare la sua attività o se rimanere attivo per eventuali chiamate future.

### 4.1.2 Architettura

Con l'istallazione di *Knative Serving*, oltre alle *Custom Resource Definition*, vengono installati diversi componenti Kubernetes:

- **Activator**: É responsabile della ricezione e della gestione delle richieste indirizzate alle Revision non attive e si occupa inoltre di inviare le metriche all'autoscaler.
- **Autoscaler**: Riceve le metriche e abilita un adeguato numero di pod per la sua corretta gestione.
- **Controller**: Ha il compito di mantenere aggiornato lo stato degli oggetti pubblici di Knative. Ad ogni Knative Service creato il controller è responsabile di convertire la *Configuration* in una serie di Revision.
- **Webhook**: Intercetta tutte le chiamate API dirette a Kubernetes nonchè tutti gli inserimenti e gli aggiornamenti delle CDRs.

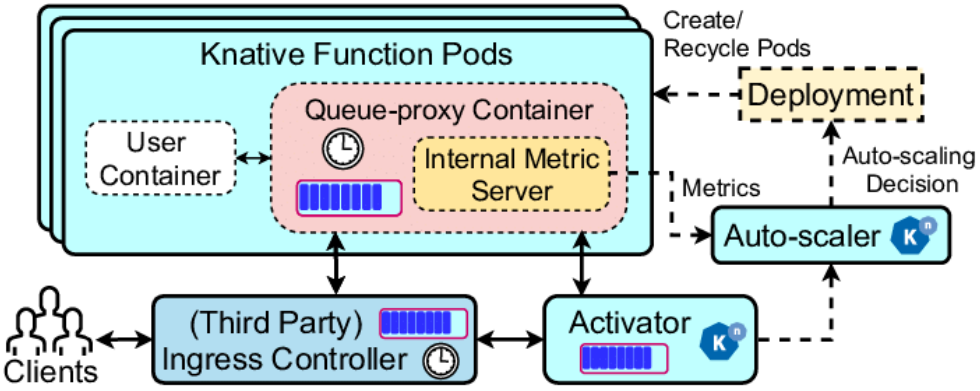



Figura 4.3: Architettura di Knative Serving

### 4.1.3 AutoScaling

Una delle principali feature di knative è lo scaling automatico di repliche della stessa applicazione per soddisfare al meglio il traffico in entrata,la particolarità rispetto ad altri framework serverless è che consente il ridimensionamento delle applicazioni fino a zero nel caso in cui non si riceve del traffico in entrata. Knative Serving abilita di default questa feature utilizzando il componente **Knative Pod Autoscaler** (KPA) il quale guarda il flusso del traffico che arriva nell'applicazione e scala le repliche in base alla metrica utilizzata. Le metriche a disposizione sono:

- **Concurrency**: Metrica che determina il numero simultaneo di richieste che possono essere processate da ogni replica.
- **RPS**: Metrica che si basa sulle richieste al secondo.
- **CPU** : Metrica basata sull'utilizza di risorse CPU.

La scelta della metrica è possibile farla inserendo l'annotazioneAUTOSCALING.KNATIVE.DEV/METRIC all'interno del file YAML che definisce un Knative Serving.

```
1  kind: Service
2  metadata:
3    name: helloworld-go
4    namespace: default
5  spec:
6    template:
7      metadata:
8        annotations:
9          autoscaling.knative.dev/metric: "concurrency"
```

Listato 4.2: Esempio Knative Serving con metrica

All'interno del file YAML di ConfigMap è possibile configurare l'autoscaler per permettere o meno lo scale a zero. Nel caso in cui è abilitato è possibile inserire un tempo di attesa, se durante questo tempo il pod non riceverà nessuna richiesta inizierà la procedura di scale a zero.

```
1  kind: ConfigMap
2  metadata:
3   name: config-autoscaler
4   namespace: knative-serving
5  data:
6   enable-scale-to-zero: "false"
7  ----
8  apiVersion: v1
9  kind: ConfigMap
10 metadata:
11  name: config-autoscaler
12  namespace: knative-serving
13 data:
14   scale-to-zero-grace-period: "40s"
```

Listato 4.3: Configurazione ConfigMap: Scale Zero e Grace-Period

Il Knative pod Autoscaler consiste in pochi componenti logici:

1. **Queue Proxy**
2. **Autoscaler**
3. **Activator**

## Queue Proxy

Si tratta di un container sidecar che viene eseguito all'interno di ogni pod di un Knative Serving. Ogni richiesta inviata a un'istanza dell'applicazione passa prima attraverso questo componente. Lo scopo di questo container è misurare e limitare la concorrenza nell'applicazione dell'utente. Se si definisce nella revision un limite a 5, il componente si assicura che non più di 5 richieste raggiungano l'istanza dell'applicazione contemporaneamente. Nel caso in cui ci siano più richieste oltre al limite queste richieste verranno messe in una coda locale.

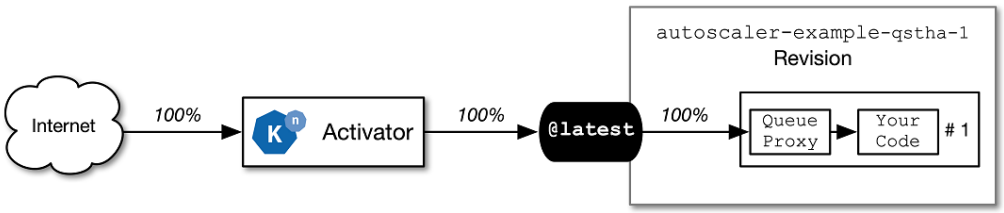



Figura 4.4: Funzionamento container Queue Proxy

## Autoscaler

É un pod che viene creato durante l'installazione di Knative Serving,al suo interno è composto da 3 componenti[18]:

- **PodAutoscaler riconciliatore**: Si assicura che qualsiasi cambiamento ai PodAutoscaler venga correttamente rilevata.
- **Collector**:É responsabile di collezionare le metriche provenienti dal componente Queue Proxy. Raccoglie le metriche interne degli endpoint dei vari pod e li riassume per ottenere una metrica rappresentativa dell'intero sistema.
- **Decider**: Ottiene tutte le metriche disponibile e decide quanti pod devono essere ridimensionati per distribuire l'applicazione in esame. Oltre a ciò, calcola quanta capacità di burst è rimasta nella distribuzione corrente e quindi determina se l'activator può essere rimosso oppure no.

### Activator

É un componente condiviso a livello globale il quale è di per sè molto scalabile. I suoi scopi principali sono il buffering delle richieste e il reporting delle metriche all'autoscaler. É principalmente coinvolto nello scaling a/da zero e nel bilanciamento del carico.Quando una revision viene scalata a zero istanze, l'activator rimane collegato alla revision per permettere che nel caso in cui arriveranno delle richieste queste verranno memorizzate all'interno del buffer dell'activator.Il componente successivamente andrà ad attivare la corretta revision in modo tale da poter inviare le metriche all'interno dell' autoscaler che agirà come un loadbalancer distribuendo il carico su tutti i pod man mano che diventeranno disponibili in modo da non sovraccaricarli in base alle impostazioni di concorrenza. A differenza del componente Queue-Proxy, l'activator invia le metriche all'autoscaler attraverso una connessione websocket per ridurre il più possibile le latenze di scalabilità.

## 4.2 Knative Eventing

Knative Eventing [19] fornisce strumenti per instradare gli eventi dai produttori di eventi ai sink(consumatori), consentendo agli sviluppatori di utilizzare un'architettura basata sugli eventi con le loro applicazioni. La particolarità di Knative eventing consiste nell'avere un disaccoppiamento tra chi produce gli eventi e chi li consuma in modo tale da avere una certa indipendenza. In questo modo, qualsiasi produttore potrà generare eventi senza che ci siano dei consumatori che ne potranno usufruire e in modo analogo qualsiasi consumatore di eventi potrà esprimere interesse per un evento o una classe di eventi, senza che ci sia effettivamente uno che li produca. Knative Eventing utilizza richieste HTTP POST standard conformi alle specifiche di CloudEvents per inviare e ricevere eventi che consentono di creare, analizzare, inviare e ricevere eventi in qualsiasi linguaggio di programmazione.La particolarità di questo componente è che consente di poter collegare altri servizi, in particolare per eseguire le seguenti funzioni:

- Creare nuove applicazioni senza modificare il produttore dell'evento o il consumatore dell'evento.
- Selezionare e scegliere come target sottoinsiemi specifici degli eventi dai loro produttori.

### 4.2.1 Componenti

Per il corretto funzionamento del framework, vengono definite alcuni componenti:

1. Event Source

2. Broker e Trigger
3. Channel
4. Event Registry

## Event source

Un *Event source* è una *CRD* di Kubernetes che viene sviluppata dall'amministratore del cluster con l'obbiettivo di fare da tramite tra chi produce l'evento e il sink. Il **sink** può essere una qualsiasi risorsa di Kubernetes incluso un *Knative Services*,un *Channel* e un *Broker*.

## Broker e Trigger

All'interno di Knative eventing sono stati introdotti due oggetti per la gestione dell'inoltro e del filtraggio degli eventi.

### Brokers

I *Brokers* sono degli oggetti definiti da *Kubernetes* come delle *CDRs* i quali definiscono un event mesh per la raccolta di un pool di *CloudEvents*. Durante la creazione di un *broker* viene fornito un endpoint ,attraverso la notazione STATUS.ADDRESS, per l'accesso degli eventi al suo interno e un *trigger* per la sua uscita. In questo modo, la risorsa che genererà l'evento potrà inviare i suoi eventi ad un broker inviando una richiesta POST allo STATUS.ADDRESS.URL.

### Trigger

Permettono agi eventi di essere filtrati in base agli attributi, in questo modo eventi con attributi particolari possono essere mandati ai subscriber che hanno suscitato maggiore interesse negli eventi con quei attributi specifici.

## Channel

*Knative Eventing* definisce un oggetto chiamato **Channel**, questo oggetto viene definito attraverso una *Custom Resource Definition* di Kubernetes con l'obiettivo di inoltrare gli eventi. Gli eventi possono essere indirizzati ai servizi disponibili all'interno del cluster oppure possono essere inoltrati ai channel utilizzando le **Subscription**.

In base al tipo del messaggio è possibile utilizzare diversi canali, i canali attualmente supportati sono :

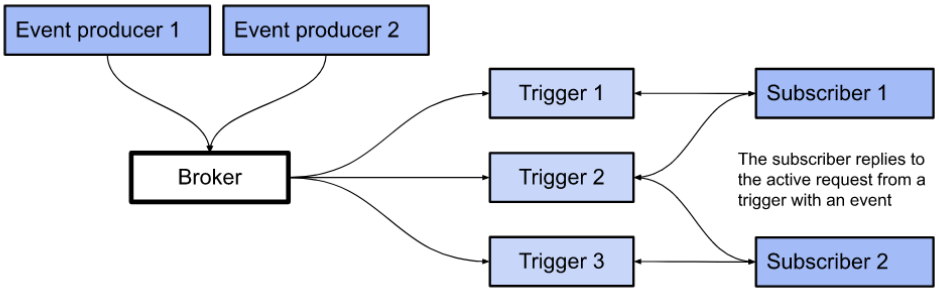



Figura 4.5: Componenti Broker e Trigger

- **GCP PubSub**: Canali definiti per *Google Cloud*
- **InMemoryChannel**: Sono canali di tipo best effort. All'interno della documentazione di Knative viene consigliato il suo utilizzo solo durante lo sviluppo in quanto non forniscono meccanismi di persistenza e non garantiscono la consegna.
- **KafkaChannel**: Canali che fanno riferimento a topic *Apache Kafka*.
- **NatsChannel**: Canali che fanno riferimento a *Streaming NATS*.

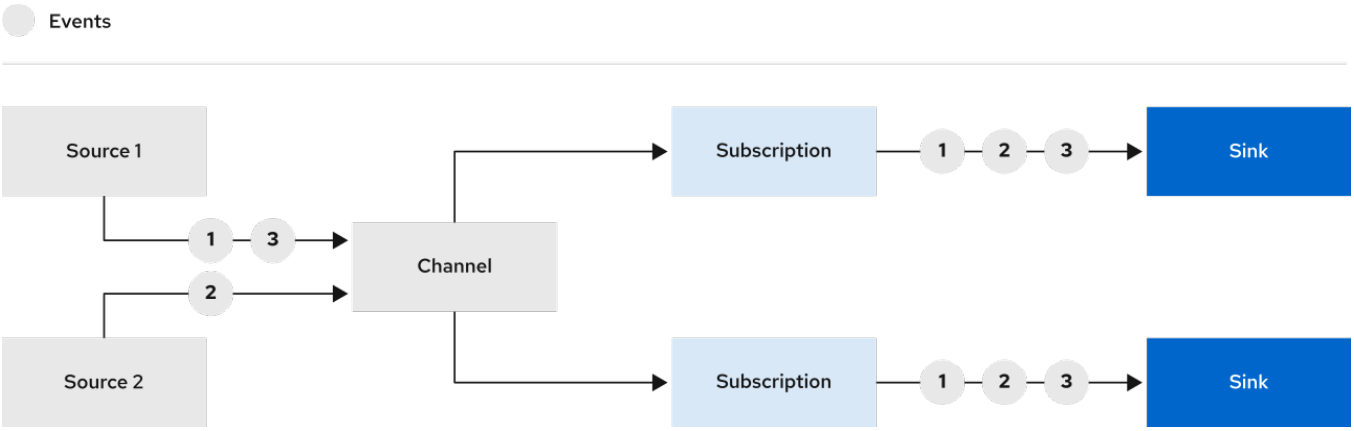



Figura 4.6: Knative Channel workflow

*Knative Eventing* fornisce di default il channel *InMemoryChannel*. Nel caso in cui si voglia creare un channel senza dover specificare quale *Custom Resource Definition* implementare è possibile creare un **Generic Channel**. Il suo utilizzo viene in nostro soccorso nel caso in cui le proprietà fornite da una particolare implementazione del canale, ordinamento e persistenza, non risultino interessanti per il suo scopo.

```
kind: ConfigMap
metadata:
  name: default-ch-webhook
  namespace: knative-eventing
data:
  default-ch-config: |
    clusterDefault:
      apiVersion: messaging.knative.dev/v1
      kind: InMemoryChannel
    namespaceDefaults:
      example-namespace:
        apiVersion: messaging.knative.dev/v1beta1
        kind: KafkaChannel
        spec:
          numPartitions: 2
          replicationFactor: 1
```

Listato 4.4: Creazione Channel

### Event Registry

Nel momento in cui si vengono a creare i *broker* e i *trigger* si avrà bisogno di un oggetto che contenga al suo interno quali tipi di eventi ogni broker potrà accettare. L'oggetto *Event Registry* consente di mantenere al suo interno un catalogo di queste informazioni, il catalogo può essere aggiornato in modo automatico o manuale. Nel caso in cui si preferisca l'aggiornamento è bene notare che viene riconosciuta solo una piccola parte di tipi di eventi come *CronJobSource*,*KafkaSource* e *AwsSqsSource*.

## 4.2.2 Flussi di eventi

Knative Eventing fornisce una raccolta di CRDs per il quale è possibile utilizzarli per definire i flussi di eventi per connettere e far cooperare diverse funzioni.

#### Sequence

Fornisce un modo per definire un elenco ordinato di funzioni che verranno invocate. Ad ogni passaggio da una funzione all'altra è possibile modificare,filtrare o creare un nuovo tipo di evento che potrà essere inoltrata alla funzione successiva.Per realizzare queste concatenazioni, Sequence si occupa di creare i necessari Channel e le necessarie Subscritpion. La figura 4.7 mostra a livello logico il suo funzionamento.

Ogni *Knative Sequence* è composto da diversi componenti:

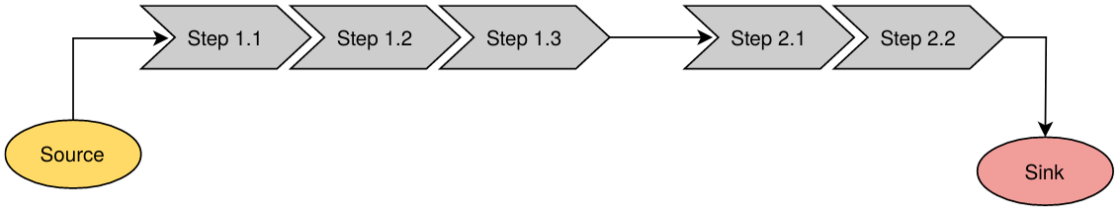



Figura 4.7: Knative Eventing- Sequence

- **Steps**: Definisce in quale ordine le funzioni verranno eseguite.
- **ChannelTemplate**: Definisce quale Template verrà utilizzata per la crezione dei Channel tra i vari step.
- **Reply**: É un parametro opzionale, utilizzato per indicare dove verranno inviati i risultati della fase finale della sequenza.

Durante l'esecuzione di una *Sequence* viene indicato uno stato:

- **Conditions**:Fornisce in modo dettagliato lo stato generale della Sequence.
- **ChannleStatus**:Trasmette lo stato dei Channel creati durante la Sequence.
- **SubscriptionStatus**: Fornisce i dettagli sullo stato per quanto riguarda le Subscription attraverso un array.
- **AddressStatus**: Fornisce i dettagli sugli indirizzi utilizzati dalle risorse Addressable.

### Parallel

Una *Parallel* come avviene per la *Sequence* viene definita attraverso una *Custom Resource Definition* con l'obiettivo di fornire una lista di branch per gli eventi. Con la sua creazione, vengono definiti una lista di branch nel quale al suo interno sono inserite una lista di *Subscriber* e un *ChannrelTemplate* per indicare il tipo di *Channel*. La figura 4.8 mostra il funzionamento logico di questo componente.

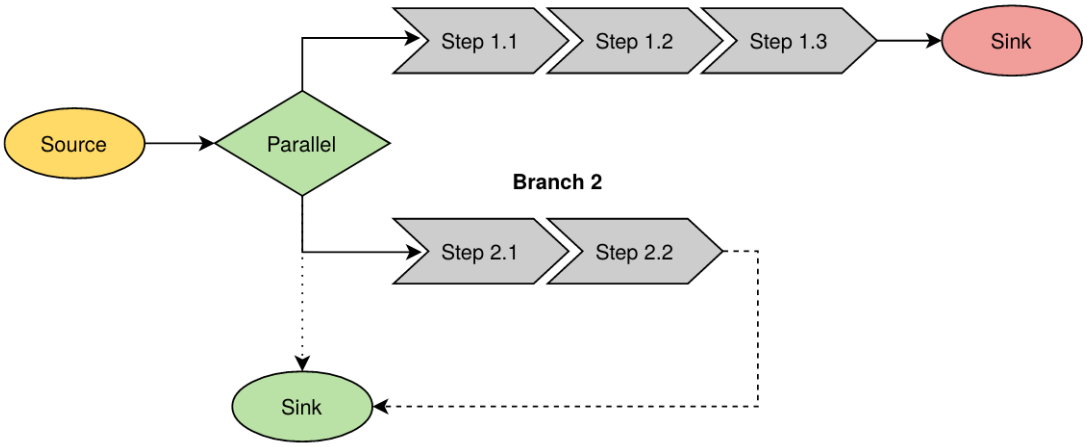



Figura 4.8: Knative Eventing- Parallel

### 4.2.3 Modalità di consegna

*Knative Eventing* dispone di tre modalità per consegnare l'evento ad un servizio.

#### Consegna Diretta

É la consegna più semplice che può realizzare *Knative Eventing*. Consiste nel mandare in modo diretto un messaggio verso un solo servizio, nel caso in cui il servizio non è disponibile verrà ritentato l'invio.

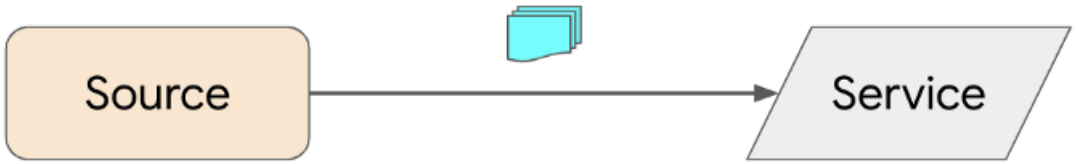



Figura 4.9: Knative Eventing- Consegna Diretta

#### Consegna Parallela

Questo tipo di consegna permette che a partire da una singola sorgente il messaggio possa andare in diversi servizi contemporaneamente grazie all'utilizzo di *channel* e *subscription*.

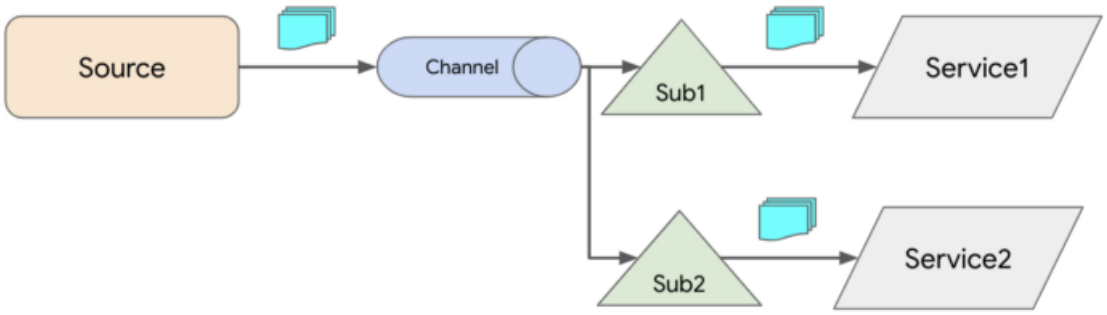



Figura 4.10: Knative Eventing- Consegna Parallela

Inoltre questo tipo di consegna permette ai servizi che ricevono i messaggi di rispondere attraverso la generazione di un nuovo evento come mostrato nella figura 4.11.

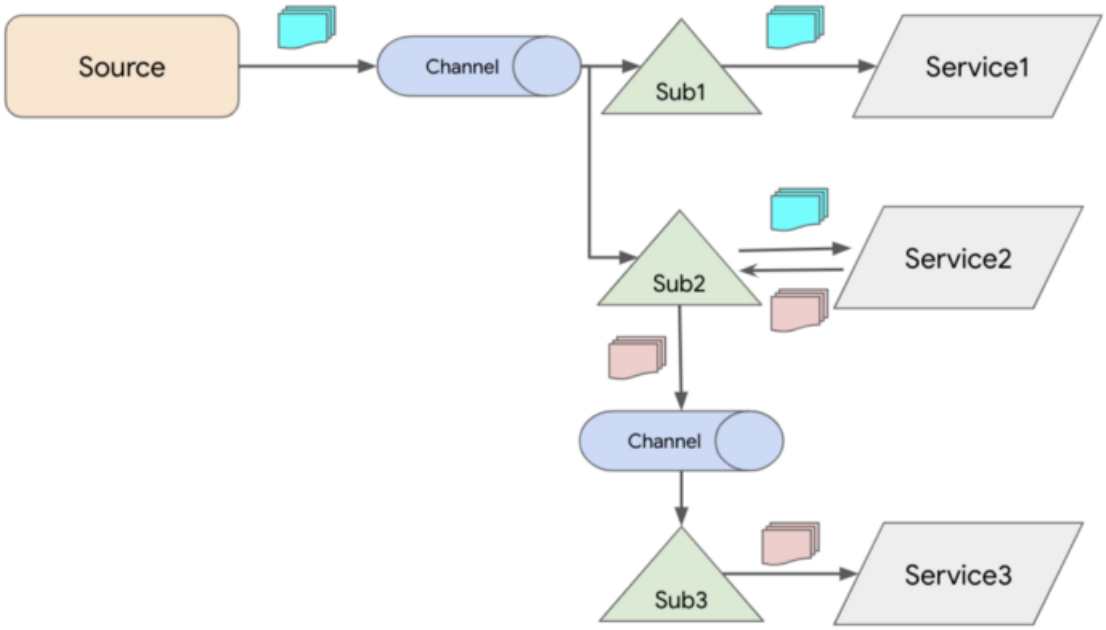



Figura 4.11: Knative Eventing- Consegna Parallela- Risposta

### Consegna attraverso Broker e Trigger

La consegna parallela risulta complessa in quanto si ha bisogno di gestire numerosi *channel*, *subscription e reply*. Inoltre, non permette di filtrare i messaggi lasciando questo compito

ai servizi. Per facilitare questo tipo di gestione vengono combinati i **Broker** e i **Trigger** in questo modo la figura 4.11 si trasforma nella figura 4.12

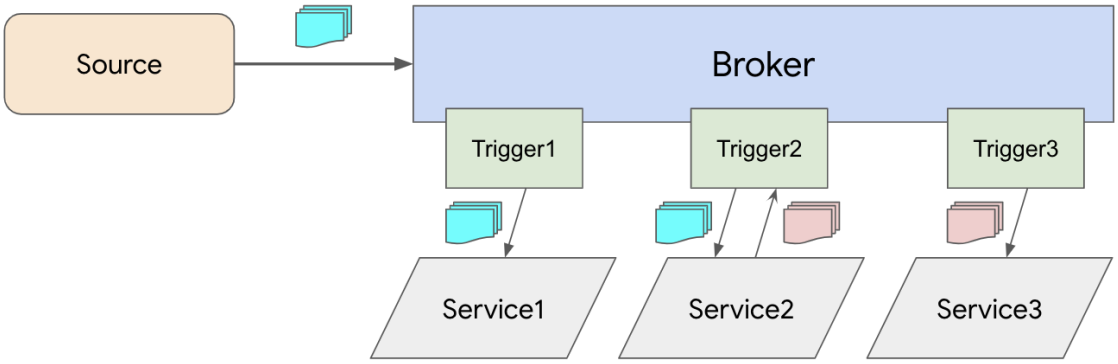



Figura 4.12: Knative Eventing- Consegna con Broker e Trigger

# Capitolo 5

# Amazon Web Services

*Amazon Web Services* [20] è una piattaforma che offre servizi di cloud computing, progettata e strutturata per poter rispondere a tutte le esigenze delle aziende operanti nel settore IT. É un cloud provider che permette di avere ogni tipologia di soluzione, dallo *IaaS* al *PaaS*, e che risolve le necessità degli sviluppatori offrendo:

- Servizi di calcolo
- Servizi di storage
- Servizi di database
- Strumenti per la migrazione
- Servizi multimediali
- Machine learning
- etc

Grazie alla sua popolarità e alla sua continua evoluzione i servizi disponibili su AWS vengono costantemente aggiornati aggiungendo nuovi prodotti quasi ogni mese. Fra i servizi più conosciuti troviamo **EC2** (*Amazon Elastic Compute Cloud*, il quale offre capacità computazionale in cloud attraverso l'utilizzo di macchine virtuali.É nato con l'obiettivo di soddisfare il bisogno di potenza di calcolo per chi non può permettersi di sostenere grandi investimenti iniziali per la costruzione di un proprio datacenter. Ogni servizio proposto da *Amazon Web Services* è altamente scalabile,veloce ed affidabile. La sua infrastruttura cloud permette di distribuire le proprie applicazioni in qualsiasi luogo grazie all'utilizzo di *Regioni,Avalability Zone,Local Zone*[21].

### 5.0.1 Regioni

In *AWS* esiste il concetto di **Regione**, esso viene inteso come un luogo fisico nel mondo nel quale al suo interno vengono clusterizzati i datacenter. Ogni gruppo di datacenter viene chiamato *Availability Zone*. Ogni Regione di AWS consiste in una serie di **AZ** isolate e fisicamente separate all'interno di un'area geografica. A differenza di altri cloud provider nel quale una ragione viene collegato ad un singolo datacenter la struttura composta in AWS permette di offrire molti vantaggi. Ogni AZ dispone di capacità di alimentazione,raffreddamento e sicurezza fisica proprie ed è connessa alle altre AZ grazie a reti ridondanti con una latenza molto bassa. La scelta di questa tipologia consente di poter distribuire la propria applicazione in diverse AZ all'interno della stessa regione per poter garantire un'alta tolleranza ai guasti.

### 5.0.2 Avalability Zone

Un'*Avalability Zone* consiste in uno o più datacenter nei quali sono provvisti di alimentazione,rete e connettività ridondanti. All'interno di una regione AWS permette si avrà un'alta disponibilità è una tolleranza ai guasti molto elevata rispetto all'utilizzo di un singolo datacenter.Tutte le AZ all'interno di una regione sono interconnesse tra di loro tramite una rete a elevata larghezza di banda e a bassa latenza tra esse ,inoltre, tutto il traffico viene crittografato. L'elevata prestazione della rete permetta di distribuire la propria applicazione su diverse AZ isolate tra loro con una distanza massima di 100Km l'una dall'altra consentendo una forte tolleranza ai guasti dovuti ad eventi naturali come blackout e fulmini e un bassa latenza.

### 5.0.3 Local Zones

Nel caso in cui alcuni utenti vogliano avere una bassa latenza tra la AZ e l'utente, *AWS* offre la possibilità di creare una **Local Zone**. É un'estensione di una *Regione* nel quale l'utente in questione si trova fisicamente. All'interno di questa zona, *Amazon* dispone di una propria connessione internet in modo tale da poter offrire una bassa latenza verso gli utenti vicini.All'interno di una *Local Zone* è possibile avviare diversi servizi tra cui: *Amazon EC2*,*Amazon ECS*,*Amazon EKS*.

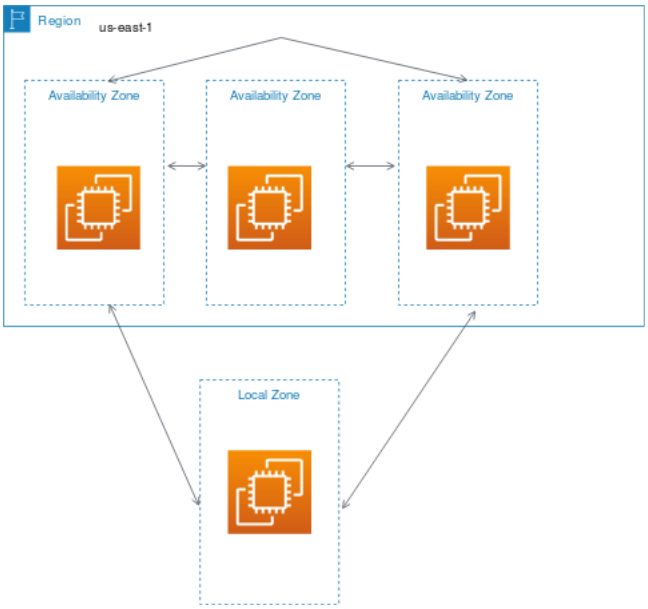



Figura 5.1: Amazon Local Zone

## 5.1 EC2

*Amazon Elastic Compute Cloud* [22] fornisce capacità di calcolo scalabile all'interno di AWS. L'utilizzo di *Amazon EC2* permette di eliminare la necessità di dover investire in hardware permettendo di concentrarsi sullo sviluppo e sulla distribuzione della propria applicazione. *Amazon Elastic Compute Cloud* è uno dei prodotti principali del servizio che fornisce Amazon, fa parte di quei servizi all'interno della categoria "*Compute*". In sostanza, si può dire che *EC2* è un servizio web che permette di rendere lo sviluppo di applicazioni più facile fornendo sicurezza e capacità di calcolo computazionale illimitata grazie alla possibilità di poter scalare l'infrastruttura in modo automatico in base alla quantità di risorse richieste per fornire le richieste ricevute. Inoltre, come tutti i servizi Amazon, consente di integrarsi con tutti gli altri servizi offerti dal cloud provider.

### 5.1.1 Amazon Machine Images

Un'*Amazon Machine Image* - *AMI* - è un template nel quale al suo interno vengono fornite tutte le informazioni necessarie per avviare una o più istanze EC2. All'interno di un template vengono fornite le seguenti informazioni:

**Root Volume:** Contiene tutte le informazioni del sistema operativo che deve adottare l'istanza EC2 e quali applicazioni devono girare al suo interno.

**Permessi di avvio:** Consente di scegliere chi può utilizzare l'immagine per poter avviare delle istanze.

**Mappatura a blocchi dei dispositivi:** Specifica quali volumi collegare all'istanza che verrà creta.

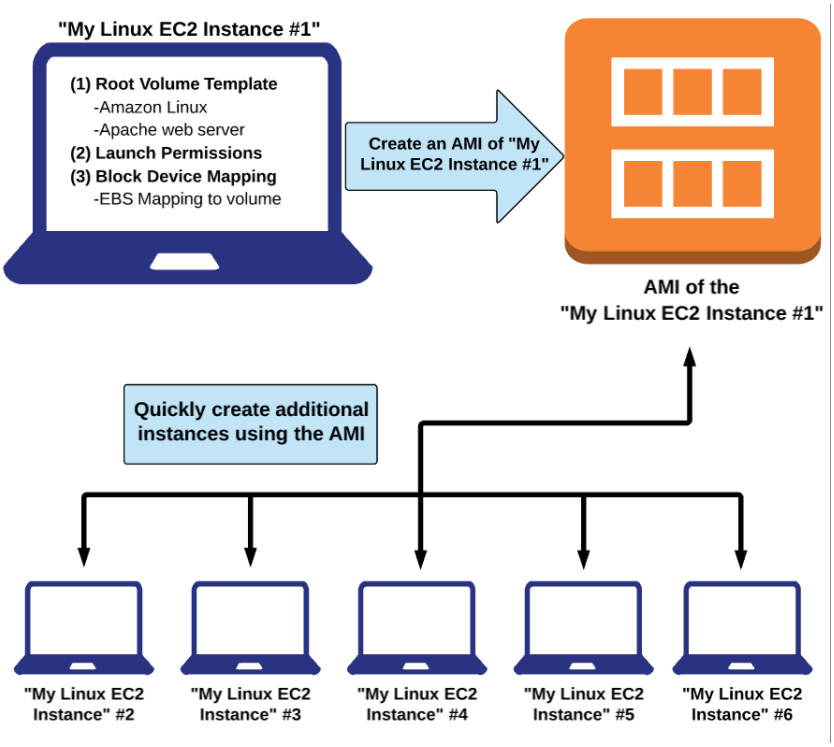



Figura 5.2: Amazon AMI: funzionamento template

**AWS** fornisce due AMI disponibili per tutti e senza costi aggiuntivi da parte del cliente: *Amazon Linux 2* e *Amazon Linux AMI*. Queste immagini consentono di avere un ambiente di esecuzione sicuro e stabile nel quale al suo interno sono disponibili i pacchetti più famosi come: *MySQL*,*PostgreeSQL*,*Python*,*Ruby*.

### Virtualizzazione

Le **AMI** consentono di far scegliere all'utente due tipi di virtualizzazione all'interno della macchina EC2: **Paravirtuale** e **Hardware virtual machine**.

## Paravirtuale

Questo tipo di approccio parte dall'abbandonare l'idea che il *Guest-OS* ,ovvero il sistema operativo installato all'interno della macchina virtuale, non possa essere modificato.Infatti,

viene esplicitamente modificato al fine di poter essere virtualizzato cambiando la sua interfaccia in modo tale da essere più facile da poter implementare. Le *Amazon Machine Image* utilizzano un bootloader chiamato **PV-GRUP** eseguito durante l'avvio della macchina EC2 nel quale carica in sequenza il kernel specificato nel template. **PV-GRUB** al suo interno contiene dei comandi che gli consentono di poter lavorare con tutte le distribuzioni Linux a partire dalla versione *10.04 LTS*. Questo tipo di virtualizzazione non consente di avere un supporto di estensioni hardware speciali come reti avanzati o elaborazioni di GPU.

### Hardware virtual machine

Questo tipo di virtualizzazione permette l'esecuzione di un sistema operativo senza che venga modificato, come può avvenire nella virtualizzazione PV, permettendo che anche quei sistemi operativi che non possono essere modificati possano essere virtualizzati. I volumi delle istanze create attraverso questo tipo di virtualizzazione vengono trattati come se fossero dischi fisici reali in modo tale da simulare un processo di avvio simile a quello che un sistema operativo normale faccia in un ambiente non virtualizzato.A differenza dei sistemi PV grazie a questo tipo di virtualizzazione è possibile sfruttare estensioni hardware esterni.

## 5.1.2 Storage

Come in tutti i cloud provider, AWS offre diversi servizi di storage per le istanze EC2 e non. Ogni opzione può essere utilizzata in modo indipendente oppure in combinazione con gli altri.I servizi di storage di AWS sono i seguenti:

- Amazon EBS
- Instance store Amazon EC2
- Amazon EFS
- Amazon S3

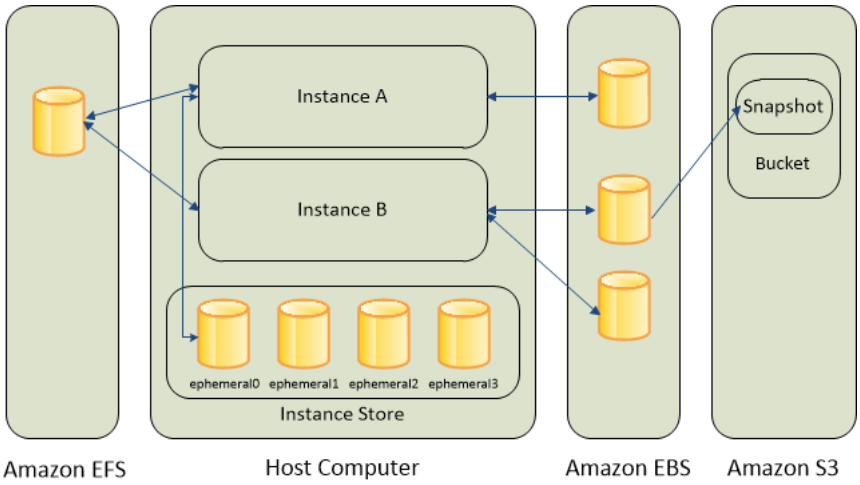



Figura 5.3: Amazon Storage

### Amazon Elastic Block Store

Amazon EBS è un servizio che permette di collegare l'istanza EC2 ad un volume formattato a blocchi.Questo volume associato ad essa rimangono attivi indipendentemente dalla durata dell'istanza EC2.Questo tipo di servizio viene utilizzata nel caso in cui le informazioni devono essere disponibili in modo rapido e con una persistenza a lungo termine. Per questo motivo sono adatti come archiviazione di un file system,database e applicazioni che richiedono una velocità di scrittura e lettura molto alta.Per permettere di ottimizzare le prestazioni e i costi sull'utilizzo di questo servizio è possibile scegliere due tipi di archiviazioni: SSD e HDD.

### Instance store Amazon EC2

Consiste sul fornire alla macchina EC2 una memoria dove allocare le risorse in modo temporaneo. Questo storage viene collocato in dei dischi fisicamente collegati alla macchina EC2. L'utilizzo di questo tipo di storage è indispensabile per quelle informazioni che cambiano molto velocemente come buffer e cache.All'interno della macchina EC2 è possibile modificare la dimensione e aumentare il numero di storage in base al tipo di istanza.
I dati vengono persi nei seguenti casi:

- Guasto del disco
- Spegnimento dell'istanza
- Ibernazione
- Terminazione istanza

Nel caso in cui l'istanza venga riavviata i dati non verranno persi.

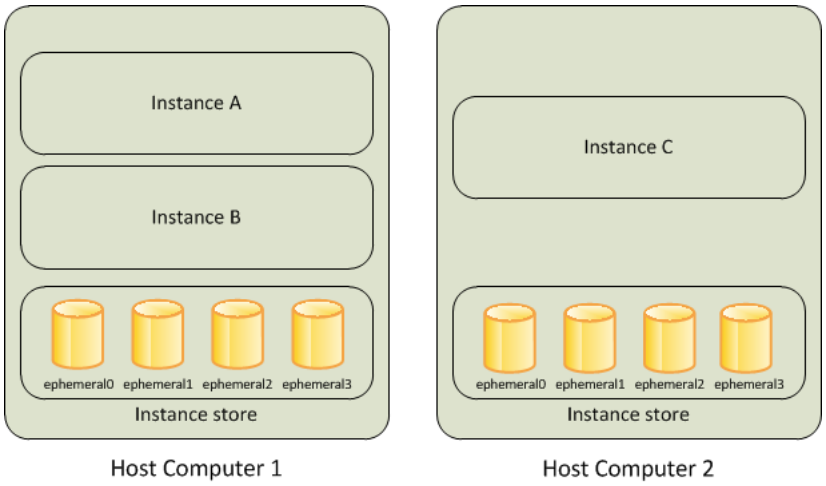



Figura 5.4: Instance store EC2

### Amazon S3

**Amazon Simple Storage Service** è uno dei servizi più utilizzati all'interno di *AWS* come servizio di storage, permette l'immagazzinamento e il recupero di informazioni di qualunque tipo, in qualunque momento e ovunque. Le caratteristiche principali che caratterizzano questo servizio sono:

- **Creazione di bucket**: I bucket sono dei container nel quale al suo interno vengono salvati i dati.
- **Storage dei dati** La quantità di informazioni che si può salvare all'interno di un *bucket* è illimitata. I dati vengono inseriti in degli oggetti, ogni oggetto può contenere un massimo di 5TB e ogni bucket può contenere un numero illimitato di oggetti.
- **Downlaod dei dati**

### Bucket

Un **bucket** è un container nel quale al suo interno vengono immagazzinati gli oggetti. Quando viene creato un bucket oltre a svolgere il compito principale ha anche diversi compiti:

- Organizzare i namespace all'interno di Amazon S3.
- Identificare chi si occupa di gestire le spese di archiviazione e trasferimento delle informazioni.
- Controllare gli accessi.

#### Oggetti

Sono dei file nel quale al suo interno vengono inseriti i metadati che descrivono le informazioni dei dati.Ogni oggetto può avere una dimensione fino a 5TB e all'interno di *Amazon S3* gli oggetti dispongono di diversi elementi:

- **Chiave**: Ad ogni oggetto viene assegnato una chiave che corrisponde al nome dell'oggetto.
- **ID Versione**: Insieme alla chiave identificano in modo univoco l'oggetto, l' *ID* viene generato da Amazon stesso quando viene aggiunto un oggetto all'interno di *Amazon S3*.
- **Valore**: É il contenuto dell'oggetto, descritto da una sequenza di byte è con una dimensione massima di 5TB.
- **Metadati**: Consiste in un set di coppie nome-valore con lo scopo di salvare le informazioni relative all'oggetto.

### 5.1.3 Amazon VPC

*Amazon Virtual Private Cloud* è un servizio che consente di eseguire i propri servizi all'interno di una rete virtuale privata fornita da Amazon. All'interno di una VPC è possibile specificare un range di indirizzi IP, subnets e route table da assegnare ai propri servizi di Amazon come ad esempio *Amazon EC2*.

#### Accesso ad Internet

Durante la creazione di una *VPC* vengono date due possibilità. La prima,chiamata **defautl VPC** consiste come dice il nome di lasciare tutto di defualt, questo tipo di approccio consente ad AWS di creare il tutto in modo automatico e facendo scegliere a lui gli IP e la subnet, la seconda, chiamata **nondefault VPC** consente all'utente di poter creare la propria *VPC* per poterla configurare come meglio creda. La differenza sostanziale tra i due consiste nell'accesso ad internet, nel caso in cui si opti per la *default VPC* ad ogni istanza creata al suo interno gli verrà assegnato automaticamente un indirizzo IP pubblico e privato mentre, nel caso in cui si scelga una *nondefault VPC*, verrà assegnato solo un IP privato e consentendo quindi la comunicazione solo all'interno della VPC a meno che non gli venga inserito un **elastic IP** durante la sua creazione. Per la comunicazione verso il mondo esterno si utilizza l'**internet gateway** che permette a chi dispone di un ip pubblico la comunicazione.

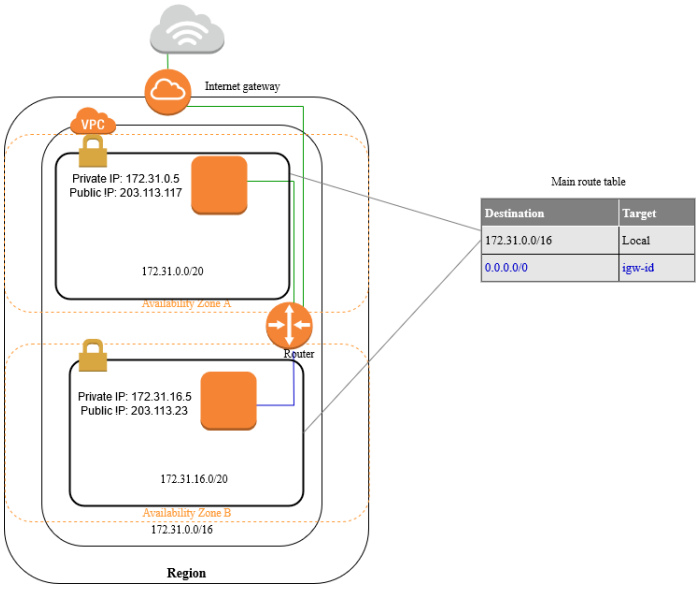



Figura 5.5: Amazon VPC

### Componenti di networking EC2

#### Elastic Network interfaces

Un'*elastic network interface* è un componente logico all'interno di un'istanza EC2 che ha il compito di collegarsi con la *VPC*. Consiste quindi in una scheda di rete virtuale nel quale al suo interno vengono definiti:

- Un indirizzo IPv4 primario privato preso fornito dal range di IP scelto per la VPC.
- Uno o più indirizzi IPv4 privati scelti sempre attraverso la VPC.
- Un indirizzo IP elastico privato.
- Un indirizzo IP pubblico.
- Uno o più indirizzo IPv6.
- Indirizzo MAC.

In base al tipo di istanza è possibile scegliere quante schede di rete associare ad essa migliorando le performance di rete e il packet rate. Ad esempio, con un'istanza di tipo *a1.large* si hanno a disposizione tre interfacce di rete e quattro indirizzi IPv4 e quattro indirizzi IPv6 per ogni interfaccia a disposizione.

#### Elastic IP

É un indirizzo IPv4 pubblico creato con lo scopo di lavorare all'interno di istanze che cambiano molto velocemente.Un *elastic IP* viene assegnato all'account che ne fa richiesta, sarà compito dell'utente assegnarlo ad un servizio o ad un'interfaccia di rete il quale permetterà di spostare tutte le informazioni contenute al suo interno da un'istanza ad un'altra con un singolo passaggio. *AWS* permette di avere per ogni account cinque elastic IP e nel caso in cui uno di questi IP non viene utilizzato deve essere ripristinato periodicamente .

### 5.1.4 Amazon EC2 Auto Scaling

Questo tipo di servizio offerto da Amazon per le macchine EC2 permette di scalare in modo automatico le istanze EC2 in base alle risorse consumate dall'applicazione. Con l'attivazione di questo servizio,completamente gratuito, le istanze EC2 che ne vogliono usufruire verranno inserite in un gruppo chiamato **Auto Scaling groups**. In ogni gruppo creatosi vengono definiti alcune specifiche come: numero minimo e massimo di istanze e capacità desiderata. Lo scaling avviene specificando le policy di scaling il quale definisce l'utente durante la creazione del gruppo.

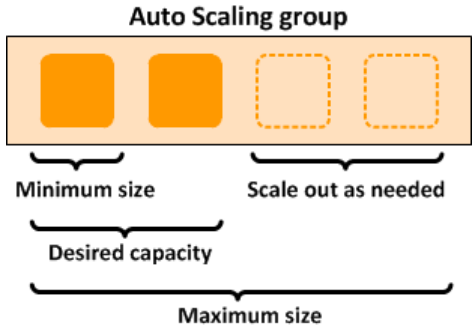



Figura 5.6: Funzionamento auto-scaling EC2

L'utilizzo di questo tipo di servizio porta numerosi benefici:

- **Tolleranza ai guasti**: L'auto scaling di EC2 riesce a capire quando la macchina ha qualcosa che non va. Nel caso in cui lo nota in automatico terminerà l'istanza per poterne lanciare una nuova. Questo tipo di politica funziona anche per quanto riguarda le *Availability Zones*, nel caso risulti non disponibile le istanze verranno eseguite in un'altra.

- **Disponibilità**: L'auto scaling assicura che l'applicazione avrà sempre le risorse a sufficienza per gestire i suoi carichi di lavoro.

- **Costo:** Avendo la possibilità di scalare in alto o in basso in base alle risorse che l'applicazione chiede si avrà una riduzione dei costi delle istanze EC2 in quanto si andranno a pagare solo quelle effettivamente utilizzate.

### 5.1.5 Pricing

*Amazon Web Services* dispone di diverse soluzioni di pricing per le istanze EC2 tra i quali troviamo :

- Istanze on demand
- Istanze Spot

#### Istanze on demand

Questo tipo di soluzione consente di pagare in base all'ora o al secondo della capacità usata per quell'istanza. La capacità di elaborazione grazie all'auto scaling di EC2 può essere aumentata e diminuita in base alle risorse chieste dall'applicazione che gira all'interno della macchina EC2. Il prezzo dell'istanza varia seconda di vari fattori:

- Regione
- Sistema Operativo
- Tipo di istanza
- vCPU
- Storage
- Prestazione di rete
- Memoria

Inoltre, oltre ai prezzi dell'istanza va aggiunto il costo dell'Elastic IP nel caso in cui se ne voglia aggiungere uno in più rispetto a quello già assegnato. Anche qui, come per le istanze, il costo viene calcolato con una tariffa oraria.

#### Istanze Spot

Questo tipo di soluzione permette di utilizzare istanze EC2 che in quel momento risultino inutilizzate da *AWS*. I prezzi risultano minori rispetto alla soluzione *on demand* in quanto si avrà uno sconto fino 90%. I prezzi variano nel tempo a seconda della disponibilità e della richiesta di quel tipo di istanza. La causa di una riduzione del prezzo è dovuto al fatto che nel caso in cui *AWS* ha bisogno di quella determinata istanza verrà fornito un avviso 120

secondi prima che la macchina venga spenta con la relativa perdita di dati. Questo deficit porta con se problemi riguardo la gestione delle applicazioni, per questo motivo le istanze spot sono consigliate per quelle applicazioni che sono tolleranti ai guasti.

## 5.2 Amazon EKS

Amazon Elastic Kubernetes Service [23] è un servizio nel quale viene data a disposizione una macchina EC2 dove al suo interno viene eseguito Kubernetes, il tutto, senza dover installare e mantenere il control plane e i nodi di Kubernetes, in quanto viene svolto in automatico da *AWS*. Amazon EKS in particolare si occupa di:

- Eseguire e scalare il control plane di Kubernetes su più zone di AWS in modo tale da garantire un'alta disponibilità.
- Ridimensiona automaticamente il control plane di Kubernetes in base al carico di lavoro, rileva e sostituisce le istanze "unhealthy".
- Fornisce gli aggiornamenti di Kubernetes in modo automatico.

Essendo un servizio di Amazon è integrato con molti dei suoi servizi per fornire scalabilità e sicurezza come Amazon VPC.

### 5.2.1 Componenti

Amazon EKS dispone di diversi componenti principali:

- EKS Control Plane
- Worker Nodes
- Fargate Profile
- VPC

#### Control Plane

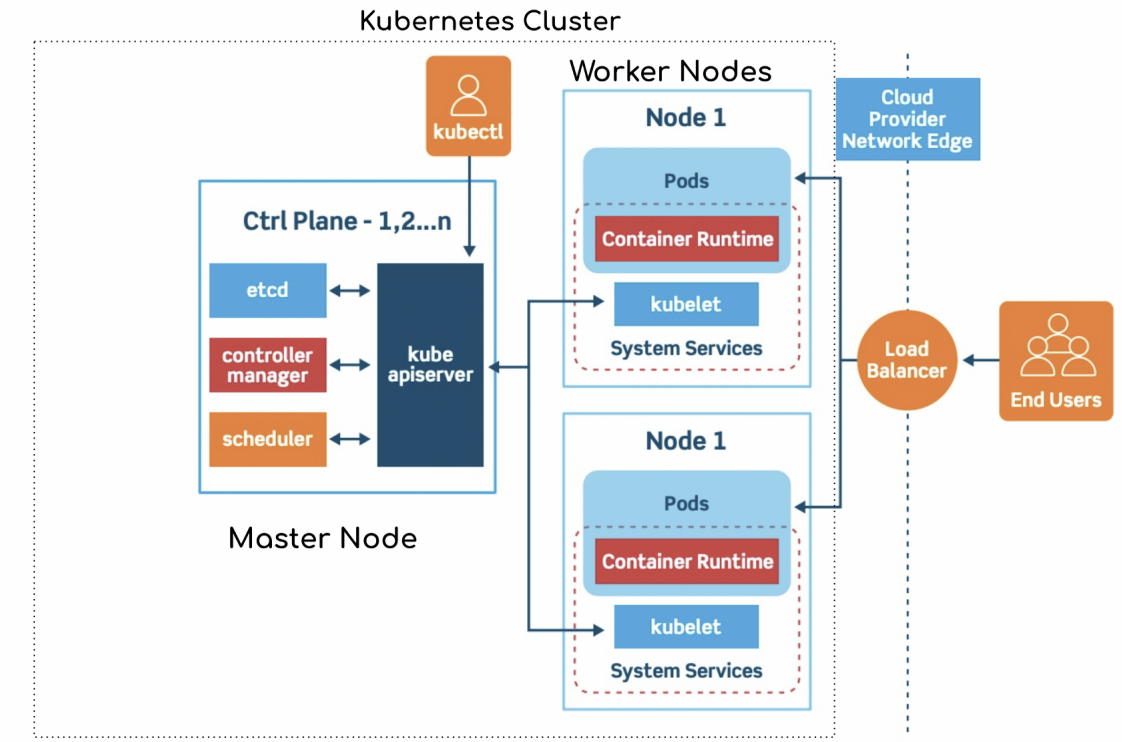
All'interno di ogni singolo cluster si trova il control plane. Il control plane detto anche *master node* consiste in una macchina EC2 il quale fornisce le API per i nodi insieme a tre istanze *etcd* che vengono eseguite su tre *Availability Zones* all'interno della regione. Il *master node* è responsabile di gestire i nodi detti *worker nodes* in modo efficiente,interagisce con loro per:

- Schedulare i pod

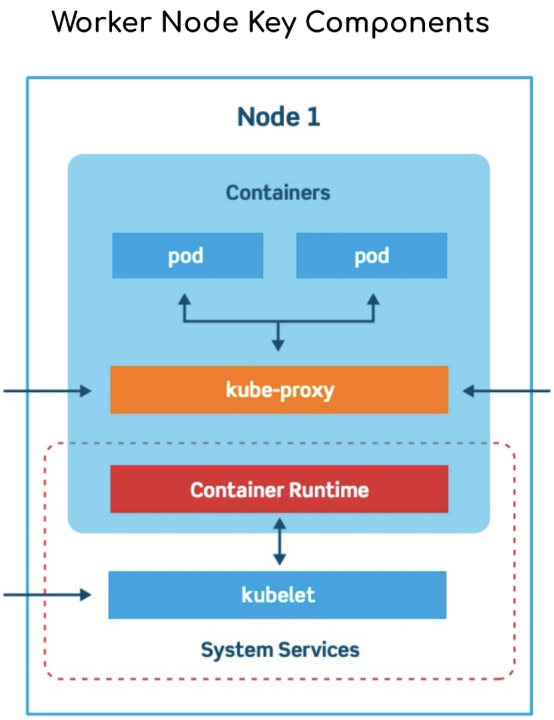
- Monitorare i pod e i nodi
- Eseguire e riavviare i pod nel caso in cui ci sia qualche problema nella loro esecuzione
- Gestire l'entrata di un nuovo *worker node* all'interno del cluster

Per poter eseguire queste funzioni il control plane in *Amazon EKS* contiene al suo interno i seguenti oggetti per ogni cluster:

- Kube-apiserver
- Kube controller manager
- kube-scheduler
- etcd



### Worker Nodes




Ogni worker node consiste in una istanza EC2 che ha il compito di avviare i pod e altri servizi gestiti da kubernetes attraverso i file *YAML* e utilizzano le API per potersi connettere con il *control plane*.Ogni istanza EC2 avrà la stessa *Amazon Machine Image* e le stesse politiche di sicurezza.

### 5.2.2 EKS Networking

Come avviene quando si crea un'istanza EC2 anche in questo caso durante la creazione di un cluster *EKS* si definisce una VPC. In questo caso specifico la macchina principale nel quale al suo interno c'è il control plane viene inserito in una VPC gestita da Amazon ed espone in modo pubblico i suoi endpoint. I nodi sono all'interno di una subnet che viene gestita dall'utente

### 5.2.3 AWS Fargate

É una tecnologia *serverless* che può essere utilizzata all'interno di *EKS* consentendo di eseguire container senza doversi occupare di tutta la gestione e configurazione delle macchine EC2.La preparazione di un'applicazione basata a container che andrà a girare attraverso

*AWS Fargate* inizia con la creazione di una **task definition** che consiste in un file di testo *JSON* nel quale si specificano ad esempio quanta CPU e memoria richiede il container.

```
{
    "family": "webserver",
    "containerDefinitions": [
        {
            "name": "web",
            "image": "nginx",
            "memory": "100",
            "cpu": "99"
        },
    ],
    "requiresCompatibilities": [
        "FARGATE"
    ],
    "networkMode": "awsvpc",
    "memory": "512",
    "cpu": "256",
}
```

A partire dalla *task definition* si andrà a creare una o più istanze chiamate **task** che andranno ad essere eseguite all'interno del cluster.Ogni *task* è isolata dal resto non condividendo CPU e memoria. *Fargate* al momento della scrittura di questa tesi consente di scegliere due tipi di sistemi operativi: *Amazon Linux 2* e *Microsoft Windows 2019 Server Full/Core*. Questo tipo di soluzione fa si che ogni volta che viene avviato una task *AWS* avvierà in modo automatico una macchina EC2 con la configurazione specificata, il **cold start** non è istantaneo ma richiederà parecchi minuti per poter essere pronto ad eseguire la task. Le ragioni di questo ritardo sono principalmente due:

- Estrazione dell'immagine Docker.
- Load Balancer

Scaricare e far partire uno o più container incide sul *cold start*, più la dimensione dell'immagine è grossa più tempo impiegherà per far partire il tutto. Inoltre, questo fattore incide anche quando il container deve smettere di funzionare, la causa è dovuto al fatto che *AWS Fargate* prima iniziare il processo di terminazione devono passare 300 secondi da quando non viene più fatta richiesta di quel determinato container ritardando ,di fatto,il tempo di avvio quando si desidera avviare un nuovo container rimuovendo quello vecchio. Questo periodo serve per il *Load Balancer* al fine di poter garantire l'integrità di tutte le istanze. L'utilizzo di *Fargate* viene dunque consigliato per determinate mansioni come:

- Carichi di lavoro che devono essere ottimizzati

- Piccoli carichi di lavoro che vengono eseguiti in modo occasionali

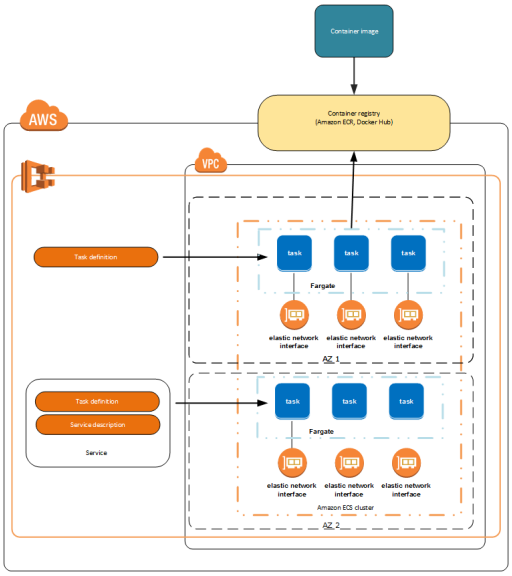



Figura 5.7: Amazon Fargate

### Pricing

Il costo relativo all'utilizzo di questo tipo di servizio si basano solo su quanta vCPU,memoria, archiviazione e sistema operativo richiede l'applicazione a partire da quanto ha inizio il download dell'immagine Docker fino al termine dell'esecuzione del container. Come in *Amazon EC2* c'è la possibilità di utilizzare delle istanze spot che permettono di avere un risparmio fino al 70% rispetto al normale prezzo.

## 5.3 AWS Lambda

É un servizio di calcolo serverless completamente diverso rispetto a quelli visti in precedenza. Similmente a *AWS Fargate*, **Lambda** [24] permette di eseguire direttamente del codice, un immagine di un container o rispondere a determinati eventi senza dover pensare alla gestione di un server o di un cluster. Il codice viene distribuito in funzioni dette **funzioni Lambda** e verranno eseguite solo se è necessario. Queste funzioni possono essere chiamati da degli eventi di altri servizi AWS oppure utilizzando un **API Gateway**

### 5.3.1 Caratteristiche

## Scaling

AWS Lambda nel momento in cui viene effettuata una richiesta per eseguire una *function* viene creata un'istanza adibita all'esecuzione di essa e rimanendo attiva per gestire ulteriori eventi. Nel caso in cui durante l'esecuzione della *function* viene richiamata *AWS* creerà una nuova istanza il quale andrà ad eseguire la stessa funzione.Nel momento in cui le funzioni non vengono più utilizzate *Lambda* incomincerà ad interrompere poco a poco quelle istanze. In ogni regione in cui la funzione Lambda viene definita si ha un un limite predefinito di *simultaneità* di 1000, il concetto di simultaneità consiste nel numero massimo di istanze che *AWS Lambda* può offrire contemporaneamente in quella regione.Nel momento in cui le richieste superano il limite di simultaneità verrano sottoposte a *throttling* e dovranno essere ripetute per essere eseguite.

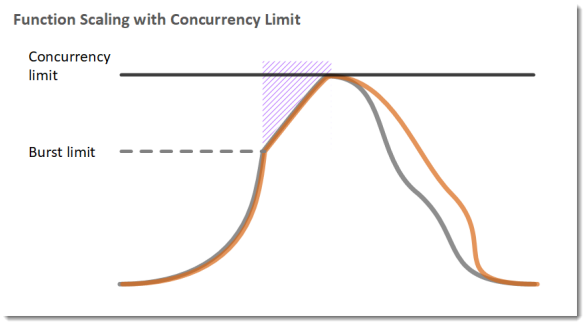



Figura 5.8: Amazon Lambda - Scaling

Attraverso l'utilizzo dell' API **Application Auto Scaling** è possibile allocare in modo automatico istanze di Lambda ancora prima che se ne siano fatte richiesta per fare in modo che siano pronte non appena ce ne fosse il bisogno ed eliminare il tempo di latenza che si avrebbe nel caso in cui si dovesse creare una nuova istanza.

## Invocazione

Nella richiesta di una funzione Lambda è possibile scegliere in che modo evocarla: sincrono o asincrono. Nell'invocazione **sincrona** bisogna attendere che la funzioni ritorni restituendo una risposta o un errore. Nella invocazione **asincrona**,Lambda inserisce l'evento scaturito da questa richiesta in una coda chiamata **event queue** restituendo subito una risposta, nel caso di una risposta negativa verrà tentata di eseguirlo con una massimo di due tentativi con un ritardo di 1 minuto tra la prima richiesta e la seconda e 2 minuti tra la seconda e la terza. Solo nel caso la funzione restituisca un errore di sistema o di *throttoling* verrano eseguiti più tentativi con un tempo massimo tra il tentativo precedente e quello successivo di 5 minuti.

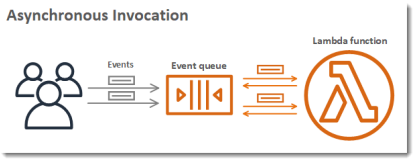



Figura 5.9: Amazon Lambda - Async

### 5.3.2 Architettura

Lambda ha a disposizioni due architetture con set di istruzioni diverse:

- arm64: Architettura ARM a 64 bit con processo AWS Graviton2.
- x86_64: Architettura x86 a 64 con processori basati su x86.

Le funzioni che utilizzano come set di istruzioni l'architettura arm64 permettono di ottenere prestazioni migliori rispetto alla funzioni basate su architettura x86. Il motivo principale è dovuto alla CPU *Graviton2* nel quale come punto di forza fornisce una cache L2 maggiore per ogni vCPU migliorando di conseguenza la latenza.

### 5.3.3 Pricing

Il sistema di pricing adottato per questo sistema si basa sul numero di richieste effettuate per le funzioni oppure sulla durata necessarie per la sua esecuzione. La durata viene calcolata a partire dal momneto in cui avviene avviata la sua esecuzione fino al momento in cui viene restituito un dato arrotondato al millisecondo più vicino.Inoltre, il prezzo si basa anche sulla quantità di memoria associata alla funzione.

## 5.4 Amazon RDS

**Amazon Relational Database Service** [25] è un servizio che consente di poter usufruire di un database relazionale in modo semplice e veloce all'interno dell'infrastruttura cloud di *AWS*. Permette di automatizzare le attività di amministrazioni in termini di tempo come,provisioning di hardware, impostazioni del database e l'applicazione di patch e backup. Al momento, questo tipo di servizio permette di scegliere sei tipi di motori: Amazon Aurora, PostgreeSQL,MySQL,MariaDB, Oracle Database e SQL Server.

### 5.4.1 Amazon Aurora

**Amazon Aurora** [26] è un database realzionale compatibile con *MySQL* e *PostgreeSQL* il quale unisce le prestazioni e la disponabilità dei database enterprise tradizionale alla semplicità e al costo ridotto dei database open source. Dal punto di vista prestazionale rispetto

ad un database MySQL standard risulta essere più veloce di cinque volte offrendo la stessa sicurezza, disponibilità e affidabilità. La scelta di questo database per la tesi è stata decisa in quanto offre la possibilità di essere serverless. **Amazon Aurora serverless** consente di dimensionare in modo automatico l'istanza del database, in questo modo consente di arrestarsi nel momento in cui non vengono effettuate delle richieste e si dimensiona in modo tale da rispondere in modo ottimale ad una richiesta.

### 5.4.2 Pricing

La possibilità di Aurora serverless di arrestarsi e riprendere l'attività nel momento in cui arriva una richiesta permette un notevole risparmio rispetto ad altre soluzioni proposte da Amazon. La tariffa, come in gran parte dei servizi, si basa su una tariffa oraria, risulta quindi utile per quelle applicazioni di cui ne usufruiscono poco durante l'arco di una giornata lavorativa.

# Capitolo 6

# Applicazione

Al fine di poter studiare il comportamento di un'architettura serverless e FaaS e con l'obiettivo di ridurre i costi all'interno di un cluster AWS, è stata realizzata un'applicazione basilare che simula una banca dati di un concessionario. L'applicazione risulta essere chiaramente un esempio dimostrativo in quanto l'obiettivo primario non è quella di creare un'applicazione completa, ma bensì lo scopo è evidenziare quali sono i vantaggi e svantaggi di una possibile applicazione messa in campo con un'architettura di questo tipo e valutarne i possibili risparmi in termini di costi con gli strumenti analizzati nei capitoli precedenti. L'applicazione è stata scritta in *Javascript* con l'utilizzo delle librerie *Express* e *Cors* per la parte server mentre *React* per quanto riguarda l'interfaccia grafica verso l'utente.

Nel percorso di questa tesi si è partiti dalla creazione dell'applicazione monolitica la quale attraverso delle richieste REST API andrà ad interagire con un database.Successivamente l'applicazione è stata divisa in diversi microservizi per poter essere implementati attraverso Knative in modo tale da poter utilizzare le risorse hardware solo ed esclusivamente nel momento in cui questi vengano chiamate. Infine, i microservizi corrispondenti alla parte backend dell'applicazione sono stati delegati ai servizi disponibili su *Amazon Web Services*.Le applicazioni sono state messe in campo utilizzando un cluster *EKS* nel quale al suo interno sono state messe in campo delle istanze *t3.medium* come worker node. La decisione di creare un'applicazione comune è dovuta al poter confrontare le stesse funzionalità anche attraverso i servizi di *Amazon*, in particolare *AWS Lambda*. Come si è visto nel capitolo precedente *Lambda* permette di eseguire delle funzioni attraverso delle immagini Docker, questa funzionalità però non permette di eseguire qualsiasi immagine, bensì permette di eseguire dei container che siano compatibili con i linguaggi supportati da esso. Esiste la possibilità di eseguire del codice non supportato all'interno di questo servizio utilizzando dei wrapper come **Bref** [27]. Questo particolare wrapper permette di far girare all'interno di una Lambda function un'applicazione come Wordpress [28] permettendo di fatto di

eseguire alcuni dei suoi componenti scritti in *php*. L'utilizzo di un wrapper come *Bref* non permette di avere una gestione totale dell'applicazione, inoltre, la possibilità di mantenere una lambda function attiva per soli 15 minuti non permette di potersi calare in un contesto reale. Per queste ragioni non è stata presa in considerazione la possibilità di utilizzare un'applicazione disponibile in rete.

## 6.1 Applicazione monolitica

L'applicazione di esempio consente di utilizzare delle chiamate API per ricevere informazioni riguardo i clienti e le macchine attualmente disponibili. Per semplicità, è stato usato un container con all'interno un database relazionale eseguito grazie a MySQL e inoltre, non sono state messe in campo metodologie per proteggere l'accesso alle API. L'applicazione è stata costruita utilizzando *React* [29] una libreria open-source Javascript sviluppata da *Facebook* con lo scopo di avere uno strumento unico per permettere la creazione di applicazioni web in modo facile e veloce, utilizzando il più possibile gli strumenti base di *HTML5* e Javascipt. Attualmente la libreria, grazie alla sua crescita molto rapida, è una delle più utilizzate per lo sviluppo web.Con queste premesse, si è creata un'applicazione monolitica la quale è vista come un unità singola e indivisibile. In particolare, l'interfaccia utente e la parte backend si trovano all'interno della stessa immagine a container. La scelta di utilizzare questo tipo di approccio per effettuare successivamente dei confronti è dovuta principalmente al fatto che nonostante l'utilizzo di questo tipo di architettura risulta essere in calo, viene reputato da numerose aziende ancora come uno standard rispetto ad un'architettura a microservizi. L'utilizzo di questa architettura pone diverse sfide in quanto qualsiasi modifica al suo interno comporta l'adattamento di tutto il codice con il rischio di compromettere il funzionamento in caso di problemi. La tabella 6.1 mostra gli endpoint disponibili in questa applicazione:

- L'endpoint GET /getClient permette di ricevere dal database i clienti che hanno acquistato all'interno del concessionario. L'API risponde con uno statusCode 200 con la lista dei clienti ricevendo i dati in formato JSON mostrato nel riquadro 6.1

- L'endpoint POST /insertClient permette di inserire un nuovo cliente all'interno del database grazie ad un form fornito nell'interfaccia grafica il quale inviare in formato JSON . L'API risponde con uno statusCode 200 nel caso in cui sia stata inserita correttamente oppure 400 nel caso di problemi.

- L'endopoint GET /getCar permette di ricevere dal database le macchine disponibili all'interno del concessionario. L'API risponde con uno statusCode 200 con la lista delle macchine.

- L'endpoint POST /insertCar permette di inserire una nuova macchina all'interno del database grazie ad un form fornito nell'interfaccia grafica. L'API risponde con uno statusCode 200 nel caso in cui sia stata inserita correttamente oppure 400 nel caso di problemi.
- L'endpoint UPDATE /updateCar/{ID} permette di aggiornare il campo "Disponibili" di uno specifico modello all'interno del database. L'API viene fornito il codice univoco del modello per permettere di poter aggiornare il giusto campo. Risponde con uno statuscode 200 nel caso sia stato eseguito correttamente oppure con 400 nel caso di problemi.
- L'endpoint DELETE /deleteCar/{ID} permette di eliminare il modello dalla lista delle macchine disponibili all'interno del database. La richiesta viene eseguita al click di un bottone all'interno dell'interfaccia grafica. All'API viene fornito il codice univoco del modello per permette di eliminare la riga corretta all'interno del database. Risponde con uno statuscode 200 nel caso in cui sia stato eseguito in modo corretto oppure 400 nel caso di problemi.

| **Richieste** | | | **Risposte** | |
|---|---|---|---|---|
| Metodo | URI | Body | StatusCode | Body |
| GET | /getClient | | 200<br>400 | Lista clienti<br>Errore |
| POST | /insertClient | Client | 200<br>400 | OK<br>Errore |
| GET | /getCar | | 200<br>400 | Lista Macchine<br>Errore |
| POST | /insertCar | Car | 200<br>400 | OK<br>Errore |
| DELETE | /deleteCar/{ID} | | 200<br>400 | OK<br>Errore |
| UPDATE | /updateCar/{ID} | Numero Macchine | 200<br>400 | OK<br>Errore |

Tabella 6.1: Endpoint API - Concessionario

Qui di seguito, sono mostrati i modelli dei dati utilizzati dall'applicazione in formato *JSON*:

```
    "ID":"Codice␣idenficativo␣del␣cliente",
    "Name":"Nome␣del␣cliente",
    "LastName":"Cognome␣cliente",
    "Phone":"Numero␣di␣telefono␣del␣cliente",
    "Email":"Email␣del␣cliente"

}
```

Listato 6.1: Modello JSON-Client

```
    "ID":"Codice␣identificativo␣della␣macchina",
    "Marca":"Marca␣della␣macchina",
    "Modelllo":"Modello␣della␣macchina",
    "Disponibilita":"Numero␣delle␣macchine␣disponibili"

}
```

Listato 6.2: Modello JSON - Car

### 6.1.1 Implementazione

Per effettuare il collegamento tra la parte backend e il container MySQL all'interno del *Dockerfile* dell'applicazione sono state inserite delle variabili ambientali per fornire i dati sul *service* creato insieme al container MySQL più il nome e i dati di accesso per il database.

```
WORKDIR /app
COPY package.json ./
COPY package-lock.json ./
COPY ./ ./
ENV MYSQL_HOST "mysql-service"
ENV MYSQL_USER "root"
ENV MYSQL_PASSWORD "root"
ENV MYSQL_DATABASE "mysqldb"
RUN npm i
CMD ["npm", "run", "start"]
```

Listato 6.3: Dockerfile-Applicazione

## 6.2 Applicazione Knative

Concettualmente, la scrittura di un'applicazione realizzata su un'architettura a microservizi e in particolare serverless consente di superare le limitazioni tipiche di uno sviluppo monolitico grazie ad un approccio modulare.Rispetto ad un'architettura monolitica ogni applicazione viene costruita unendo diversi moduli indipendenti tra di loro che se modificate, non andranno ad interferire con il funzionamento complessivo del software. In questa tesi, come è stato ampiamente descritto, l'obiettivo è cercare di ridurre i costi al minimo attraverso l'utilizzo di Knative e successivamente integrandolo con i servizi AWS.Tipicamente, durante la scrittura di un'applicazione serverless vengono formulate diverse ipotesi nel quale si immaginano diversi scenari sul suo utilizzo,nel nostro caso, si ipotizza che l'applicazione in esame venga utilizzata per brevi periodi e che non tutti i servizi vengano utilizzati nello stesso momento. La possibilità che offre Knative permette quindi di implementare numerosi Knative Services senza che questi consumino quantità di CPU e RAM durante il loro stato di riposo in quanto Knative nel momento in cui il Knative services non verrà più richiesto scalerà a zero i pod. Con l'obiettivo di ridurre al minimo il consumo di CPU e RAM da parte dell'applicazione è stata richiesta una parziale riscrittura del codice, in particolare sono state effettuate le seguenti scelte architetturali come è possibile vedere nella figura 6.1:

- É stato creato un *Knative Services* chiamato frontend il quale se chiamato fornisce l'interfaccia grafica per poter interagire con il Database.
- L'endpoint GET /getClient è stato trasformato in un *Knative Service* che viene attivato nel momento in cui si effettua la chiamata */getClient* nel Knative Service frontend.
- L'endpoint POST /insertClient è stato trasformato in un *Knative Service* che viene attivato nel momento in cui si effettua la chiamata */insertCar*, riceve dal servizio frontend in accordo con il modello JSON 6.1 i dati relativi ai clienti.
- L'endpoint GET /getCar è stato trasformato in un *Knative Service* che viene attivato nel momento in cui si effettua la chiamata */getCar* nel Knative Service frontend.
- L'endpoint POST /insertCar è stato trasformato in un *Knative Service* che viene attivato nel momento in cui si effettua la chiamata */insertCar*, riceve dal servizio frontend in accordo con il modello JSON 6.2 i dati relativi alla macchina.
- L'endpoint DELETE /deleteCar/{ID} è stato trasformato in un *Knative Service* che viene attivato nel momento in cui si effettua la chiamata */deleteCar/{ID}*, riceve

dal servizio frontend il parametro ID per poter eseguire la query per eliminare la riga sul database.

- L'endopoint UPDATE /UPDATECAR/{ID} è stato trasfrormato in un *Knative Service* che viene attivato nel momento in cui si effettua la chiamata */updateCar/{ID}*, riceve da parte del servizio frontend l'ID della macchina come parametro e nel body il numero di macchine disponibili nuovo.

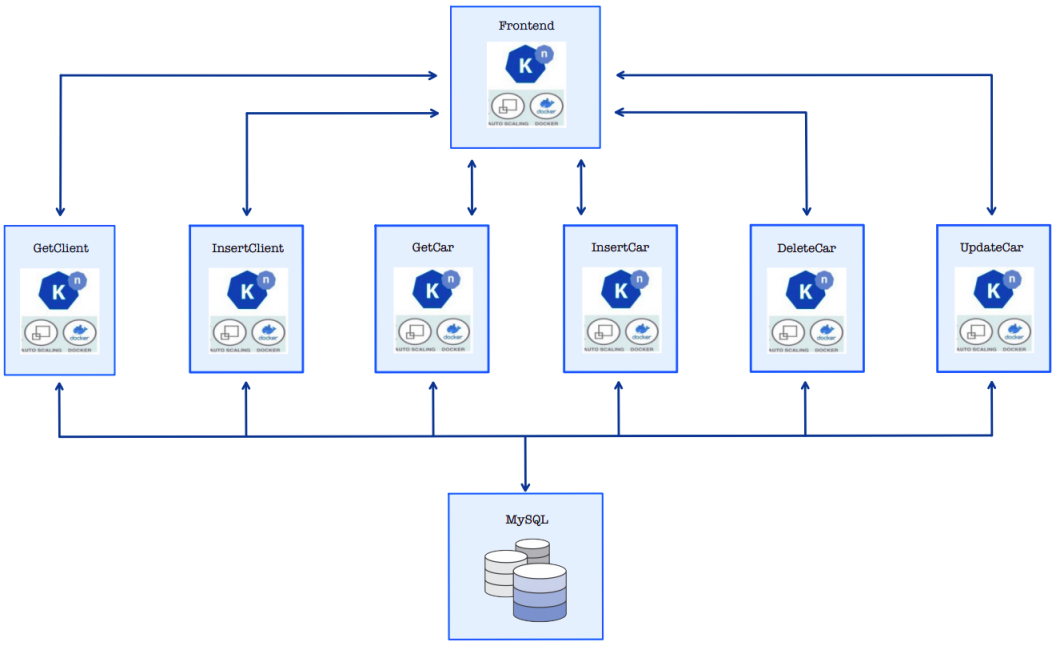



Figura 6.1: Applicazione implementata con Knative

La scelta di questo tipo di architettura permette di sfruttare appieno il funzionamento di *Knative* in particolare il suo autoscaling in quanto ogni servizio verrà gestito in modo tale da spegnersi non appena non riceverà più richieste e andando a liberare delle risorse hardware.

### 6.2.1 Implementazione

Durante la fase di transizione da un applicazione monolitica ad un'applicazione serverless gli $n$ endpoint sono stati suddivisi in $n$ moduli, ciascuno responsabile del singolo endpoint che si interfacciava con la sua API REST. Si è ipotizzato che, dato $M$ il quantitativo di ram utilizzata per l'applicazione monolitica, è stata suddivisa tra i vari Knative services in quantitativi minori tali che:

$$\sum_{k=1}^{N} m_{\mathrm{k}} = M$$

Per quanto riguarda il consumo di CPU a livello computazionale dato un modulo dell'applicazione monolitica, come ad esempio la richiesta di */getClient*, questo è stato riportato in modo uguale anche per la controparte serverless in quanto il costo computazionale del servizio entra in gioco solo quando viene effettuata una richiesta al database indipendentemente dal tipo di architettura che si è scelto.
I servizi creati da Knative sono richiamabili attraverso delle richieste HTTP su un dominio. Per questo tipo di scenario è stato utilizzato il servizio **sslip.io** che ha il compito di fare da DNS per assegnare ai vari servizi un URL temporaneo definito come <nomeServizio>.<namespace>.<indirizzo IP>.sslip.io. Per tutelarsi dalla possibilità di attacchi esterni su alcuni servizi è possibile impostare l'etichetta **serving.knative.dev/visibiliy** con il valore **cluster-local** per rendere visibile il *Knative Service* solamente all'interno del cluster. In questo esempio, solamente il servizio *frontend* è stato reso pubblico mentre per gli altri è stata inserita l'etichetta *cluster-local*.

```
1 kind: Service
2 metadata:
3  name: get-client
4  labels:
5    app: get-client
6    networking.knative.dev/visibility: cluster-local
7 [...]
```

Listato 6.4: Dockerfile-Knative Services

La costruzione dell'applicazione con i *Knative Service* etichettati *cluster-local* comporta il dover esporre in modo manuale le API. La libreria React in un contesto development necessita di un proxy per poter funzionare in modo corretto, per questo motivo è stata implementata la libreria *http-proxy-middleware* all'interno del *Knative service* frontend per indirizzare opportunamente il traffico verso il corretto servizio come mostrato nell'esempio sottostante.

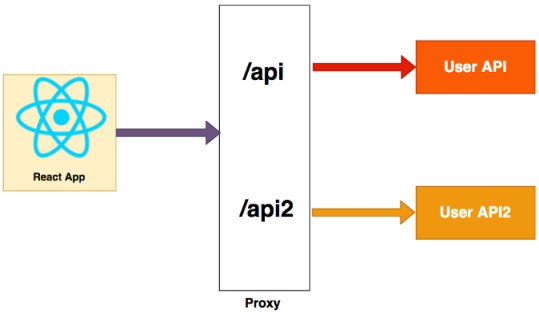



Figura 6.2: Esempio Proxy

## 6.3 Integrazione con i servizi di AWS

Come si è potuto descrivere nel capitolo 5 la piattaforma Amazon offre numerosi servizi che possono aiutare durante lo sviluppo di un'applicazione, in questa sezione si cercherà di analizzare come sono stati introdotti alcuni dei servizi offerti all'interno di questa applicazione di esempio. L'applicazione è stata implementata facendo cooperare *Knative* con i servizi disponibili all'interno di *Amazon Web Serivces*. La scelta di utilizzare come piattaforma cloud *AWS* è dovuta perchè risulta uno dei maggiori cloud provider utilizzate dalle aziende IT in quanto offre numerosi vantaggi come:

- Facilità d'uso
- Convenienza
- Affidabilità
- Sicurezza

A livello di codice, l'applicazione è stata rivista in parte, in particolare per quanto riguarda la parte backend. Tenendo sempre in conto l'obiettivo di ridurre i costi sono state eseguite le seguenti implementazioni:

- Il *Knative service* che si occupava di fornire l'interfaccia grafica è stata modificata in modo tale da poter interagire con i servizi di AWS.
- Il *Knative service* che si occupava di rispondere alla richiesta /GETCLIENT è stata trasformata in una Lambda function.
- Il *Knative service* che si occupava di rispondere alla richiesta /INSERTCLIENT è stata trasformata in una Lambda function.
- Il *Knative service* che si occupava di rispondere alla richiesta /GETCAR è stata trasformata in una Lambda function.

- Il *Knative service* che si occupava di rispondere alla richiesta /insertCar è stata trasformata in una Lambda function.
- Il *Knative service* che si occupava di rispondere alla richiesta /deleteCar è stata trasformata in una Lambda function.
- Il container MySQL è stato eliminato per utilizzare il servizio Amazon Aurora Serverless

## Implementazione

### 6.3.1 Amazon Aurora Serverless

Come viene spiegato nella sezione 5.4.1, *Aurora serverless* permette di gestire un database relazionale attraverso ad esempio l'engine MySQL in modo facile e veloce. L'utilizzo di questo servizio a discapito di altri come *Dynamo DB* o *RDS* risiede sul mantenere un database relazionale come nelle altre architetture ma, che permetta anche di ridurre i costi grazie alla sua modalità serverless. Durante la creazione del servizio è possibile scegliere oltre al tipo di engine da utilizzare (*MySQL* o *PostGreeSQL*) anche dopo quanto tempo il database dovrà smettere di funzionare nel caso in cui non ci siano delle connessioni attive. Nel nostro scenario, è stato utilizzato l'engine MySQL e impostato un tempo di inattività a 5 minuti ( il minimo ). Successivamente è stato utilizzato il servizio per la creazione di una secrets per poter salvare in modo sicuro i dati per poter accedere come admin all'interno del database.

### 6.3.2 Lambda Function

*AWS Lambda* permette di eseguire codici di una qualsiasi applicazione o servizio back-end senza dover effettuare il provisioning o gestire i server. La scelta di integrare questo tipo di servizio all'interno di questa tesi è stata fatta in quanto Lambda può essere visto come un'alternativa a Knative ed inoltre ha un tempo di cold start quasi nullo ad un costo molto basso, basato non sulla potenza di calcolo per la corretta esecuzione ma quanto sul numero di functions eseguite.Per potersi interfacciare con il database implementato con Aurora serverless è stata utilizzata la libreria **RDS DATA Service**. La particolarità di questa libreria consente di effettuare delle chiamate HTTP verso il database. La scelta è dovuta principalmente ad un risparmio in termini di costi. Effettuando una connessione tipica come ad esempio con il comando *mysql.Connection()*, il collegamento resterebbe attivo fino a quando la Lambda function non smetterà di funzionare ( 15 minuti ) facendo, di fatto, aumentare i costi inutilmente in quanto il database resterà attivo. La possibilità di utilizzare delle richieste HTTP permette di ridurre il tempo di connessione al database solo

per il suo effettivo utilizzo e di conseguenza ridurre i costi di mantenimento del database da parte dell'azienda. Per poter eseguire una query, all'interno della Lambda function vengono forniti i seguenti parametri:

- secretArn: Contiene le credenziali di accesso al database.
- resourceArn:Contiene l'ARN del database su cui si andrà a fare l'accesso.
- sql:Contiene la query da eseguire.
- database:Contiene il nome del database.

Se da un lato questo tipo di implementazione permette di effettuare il collegamento al database solo quando è effettivamente necessario, dall'altro lato si hanno due principali svantaggi:

1. Nel caso in cui il servizio Aurora Serverless non risulti attivo al momento della richiesta da parte di una Lambda function esiste la possibilità che non venga ricevuta alcuna risposta in quanto il tempo di cold start risulta abbastanza alto. Per ovviare questo problema, è stato impostato un tempo di timeout di 1 minuto e, nel caso in cui non si riceva una risposta da parte del database, verrà ritentata un'altra connessione.

2. Utilizzando una connessione attraverso delle richieste HTTP si ha un limite di quanti dati si possano ricevere come risposta da parte del database. In Aurora serverless la dimensione massima di dati che può inviare come risposta risulta essere di 1 MB, questo limite,nel nostro scenario di esempio, non viene superato ma, in caso di database di grosse dimensioni, dovrà essere considerato se la scelta di utilizzare questo tipo di servizio potrà essere una soluzione vantaggiosa o no.

### 6.3.3 API Gateway

Amazon API Gateway è un servizio AWS per la creazione, la pubblicazione, la gestione, il monitoraggio e la protezione di API REST, HTTP e WebSocket a qualsiasi livello.La sua funzione permette di essere una "porta principale" per poter accedere ai servizi offerti da AWS. Nel nostro scenario di esempio è stato utilizzato come "trigger" per poter richiamare le Lambda function in modo sicuro. Sono state create due API distinte:

- É stata creata l'API **/Client/** che accetta le richieste API REST fornite dall'applicazione che riguardano la gestione del cliente. Con la sua creazione sono stati creati gli endpoint */getClient* e */insertClient* i quali si interfacciano con le loro rispettive Lambda function.

- É stata creata l'API **/Car/** che accetta le richieste API REST fornite dall'applicazione che riguardano la gestione delle macchine disponibili. Con la sua creazione sono stati creati gli endpoint */getCar*,*/insertCar*,*/deleteCar/{:ID}* e */updateCar/{:ID}* i quali interfacciano con le loro rispettive lambda function.

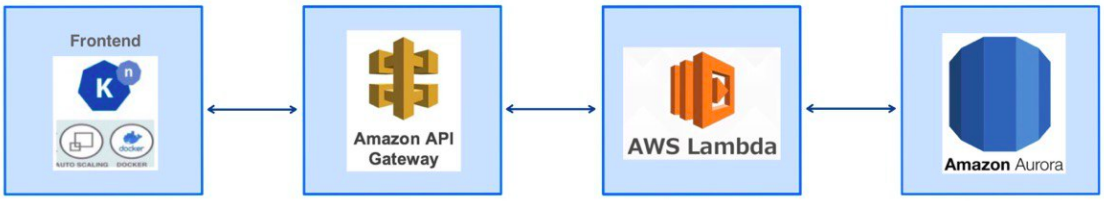



Figura 6.3: Implementazione applicazione AWS

# Capitolo 7

# Test

Per poter valutare le performance dell'applicazione delle architetture descritte nel capitolo precedente, i test sono stati effettuati utilizzando l'applicativo *Apache JMeter* in modo tale da poter simulare picchi di lavoro all'interno dell'applicazione. Per la raccolta dati sono stati utilizzati due strumenti **Prometheus** e **Grafana**. **Prometheus** [30] è una piattaforma di monitoring che offre un modello di dati con serie temporali che può essere interrogata con un linguaggio flessibile, noto come *PromQL*.Questo tipo di applicativo, risulta particolarmente adatto nel momento in cui si vogliono visualizzare delle statistiche globali all'interno del sistema. **Grafana** [31] è una piattaforma open-source con lo scopo di analizzare e monitorare qualunque tipo di base di dati come ad esempio quelle salvate da *Prometheus* offrendo numerosi grafici per facilitare la visualizzazione dei dati.

Le figure che verranno mostrate sono state prese direttamente da grafana nel quale vengono riportati gli utilizzi delle risorse hardware, in termini di CPU e memoria RAM delle diverse funzioni analizzate.É bene notare che nel mondo del cloud e in particolar modo in Kubernetes, la misura dell'utilizzo di CPU viene effettuata in *unità cpu* il quale corrisponde ad una *virtual CPU o virtual core*. Nel caso di applicazioni che non sfruttano appieno tutta la potenzialità della CPU può essere utile utilizzare dei sottomultipli del core. Nel nostro caso è stato utilizzato il millicore, abbreviato come *mcore*, il quale indica la millesima parte di un *core*.

## 7.1 Confronto Applicazioni a riposo

Al fine di poter confrontare le applicazioni realizzate, è necessario valutare quali siano le risorse impiegate nello stato di riposo, in particolare, i dati raccolti in un periodo dove non sono state ricevute ed elaborate delle richieste.

### 7.1.1 Applicazione monolitica

Nella figura 7.1 e 7.2 vengono mostrate le risorse utilizzate dall'applicazione per quanto riguarda CPU e memoria. L'attività in termini di CPU risulta relativamente bassa mentre per quanto riguarda il consumo di RAM risulta decisamente alto dovuto anche al database al suo interno.

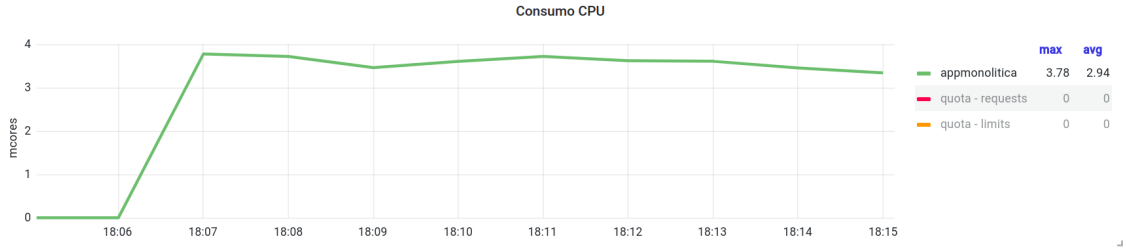



Figura 7.1: Applicazione Monolitica Riposo - CPU

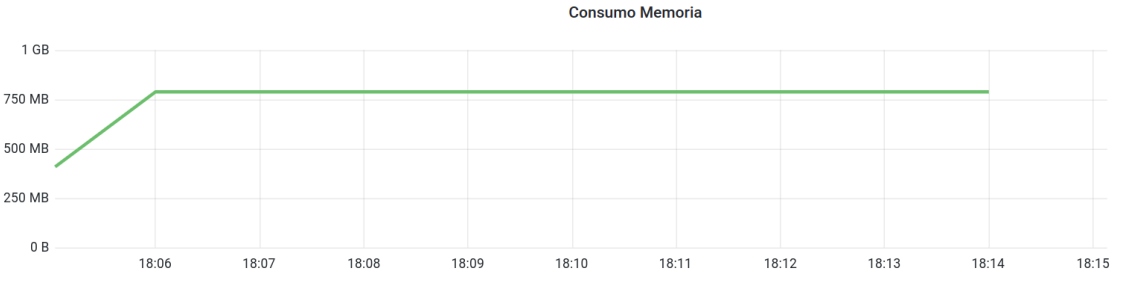



Figura 7.2: Applicazione Monolitica Riposo - Memoria

### 7.1.2 Applicazione Serverless

Nonostante nella nostra architettura è stato scelto di utilizzare lo "scale to zero" offerto da Knative per poter sfruttare il meno possibile le risorse. Si è voluto confrontare un *Knative Service* durante lo stato di riposo, è stato dunque modificato il file YAML contente l'interfaccia grafica dell'applicazione inserendo l'annotazione AUTOSCALING.KNATIVE.DEV/MINSCALE: "1". Come è possibile vedere dalle figure successive il consumo delle risorse lato CPU risulta più alto rispetto all'applicazione monolitica in quanto Knative, responsabile della messa in campo di un *Knative Service*, implementa altre funzionalità rispetto ad un deployment

Kubernetes normale. É possibile notare infatti, dalle figure 7.3 e 7.4 che il container *queue-proxy* implementato automaticamente da Knative come un sidecar, non risulti trascurabile nonostante nel container *frontend* venga eseguito il framework React.

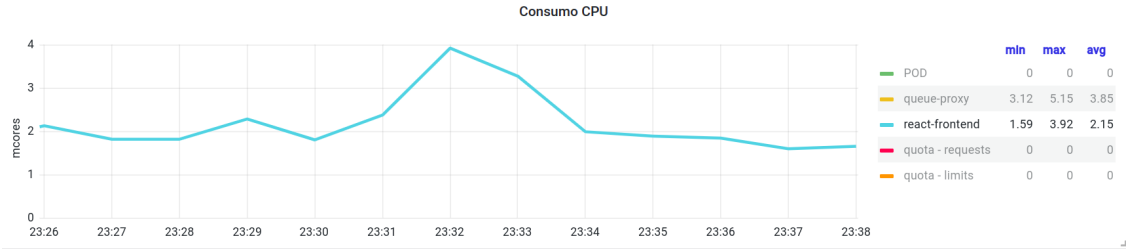



Figura 7.3: Container Frontend Riposo - CPU

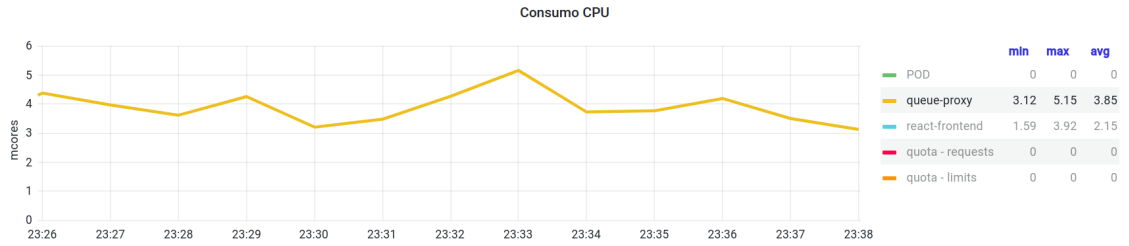



Figura 7.4: Sidecar Queue Proxy Riposo- CPU

Per quanto riguarda il consumo di memoria la situazione non è analoga in quanto rispetto ad un consumo piuttosto elevato del container *frontend* di 350MB il consumo del sidecar *queue-proxy* risulta piuttosto trascurabile come si può vedere dalla figura 7.5

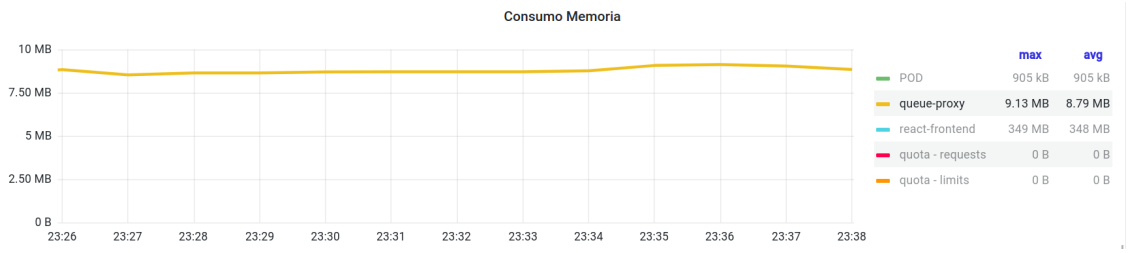



Figura 7.5: Sidecar Queue-Proxy Riposo - Memoria

Per completezza sono stati raccolti anche i dati del *Knative service* responsabile di gestire l'endpoint della eliminazione di una macchina dal catalogo, in modo analogo, si ottengono risultati simili per l'endopoint dell'update del catalogo. In questo caso, essendo più leggero dal punto di vista computazionale, il container *queue-proxy* risulta più importante lato CPU:

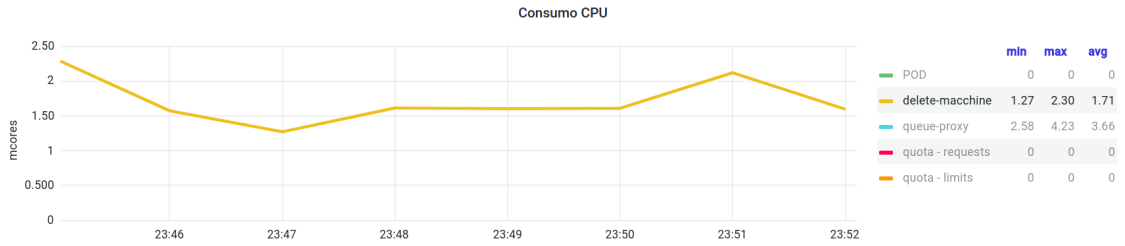



Figura 7.6: Container DeleteCar Riposo - CPU

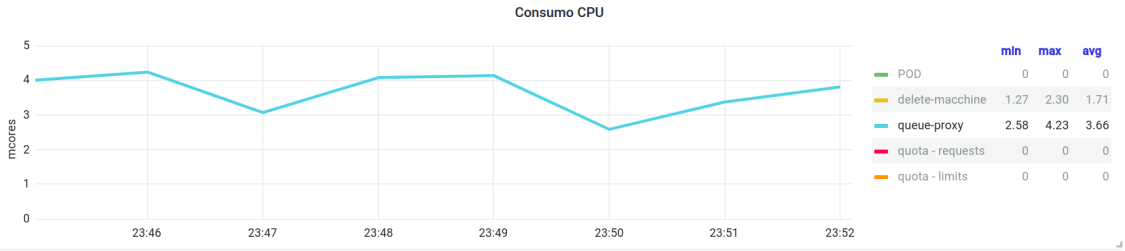



Figura 7.7: Sidecar Queue-Proxy DeleteCar Riposo - CPU

### 7.1.3 Knative

Nel momento in cui si installa un framework on-top di Kubernetes, come Knative, risulta utile avere dei dati che mostrano il consumo lato CPU e RAM. Knative come abbiamo potuto vedere nel capitolo 4 offre di base delle politiche di autoscaling, messe in campo come sidecar chiamati **Queue-proxy** associati ad ogni deployment. Di contro, i benefici portano ad una complessa architettura risultando pesante nell'utilizzo delle risorse inoltre, la messa in campo di codice risulta più complessa e meno immediata in quanto bisogna effettuare le build delle immagini Docker prima di poter effettuare il deployment. Le figure 7.8, 7.9, 7.10 e 7.11 mostrano i consumi dei namespace creati durante l'installazione di Knative, **Knative Serving** e **Knative Eventing**, creati ed utilizzati per una corretta esecuzione del framework.

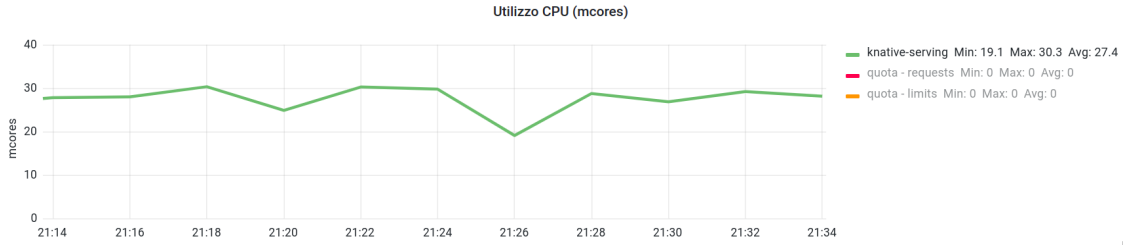



Figura 7.8: Knative Serving - CPU

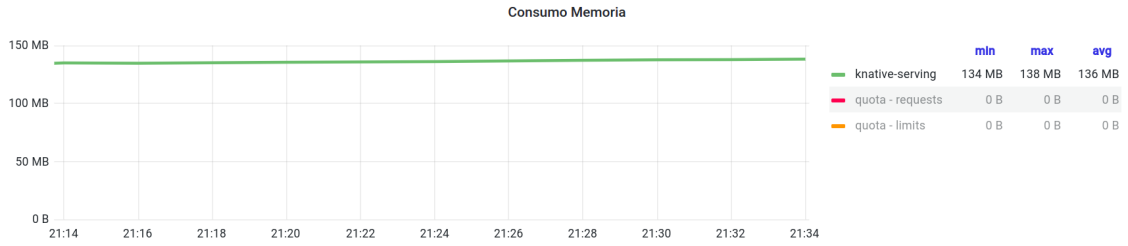



Figura 7.9: Knative Serving - Memoria

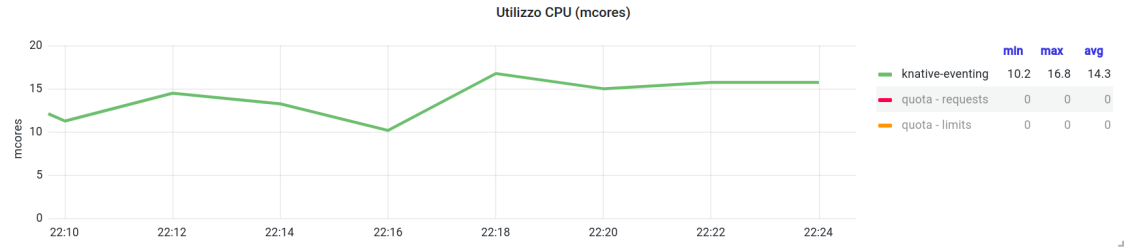



Figura 7.10: Knative Eventing - CPU

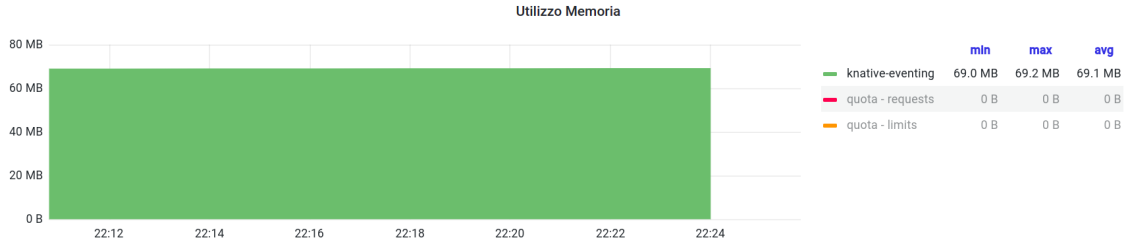



Figura 7.11: Knative Eventing - Memoria

## 7.2 Inserimento Singolo

L'obiettivo di questo test è valutare la risposta delle applicazioni create in seguito all'inserimento di un singolo dato. Prendendo in considerazione il caso d'uso di un concessionario è possibile ipotizzare che durante l'intero arco di giornata vengano fatte poche richieste di aggiornamento del catalogo delle macchine all'interno del database.

### 7.2.1 Applicazione monolitica

Per quanto riguarda l'inserimento singolo utilizzando l'applicazione monolitica, non risulta significativo l'utilizzo delle risorse utilizzate rispetto al suo stato di riposo riportati in figura 7.1 e 7.2.

### 7.2.2 Applicazione serverless

Come è possibile vedere dalle figure 7.12 e 7.13, l'applicazione nel complesso ha un consumo significativo rispetto all'applicazione monolitica. La causa è dovuta principalmente alla peculiarità di Knative di creare diversi Knative services in base alla richiesta, nel nostro caso ricevendo la chiamata */insertCar* inizializza due pod, la parte frontend e la parte backend responsabile di elaborare la richiesta.

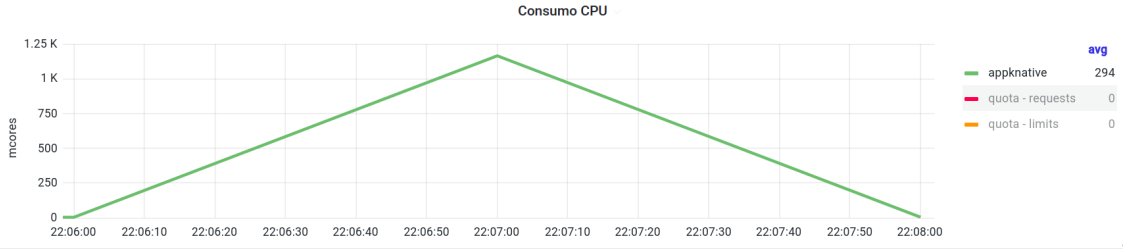



Figura 7.12: Knative service InsertCar CPU - Risorse utilizzate per l'inserimento singolo

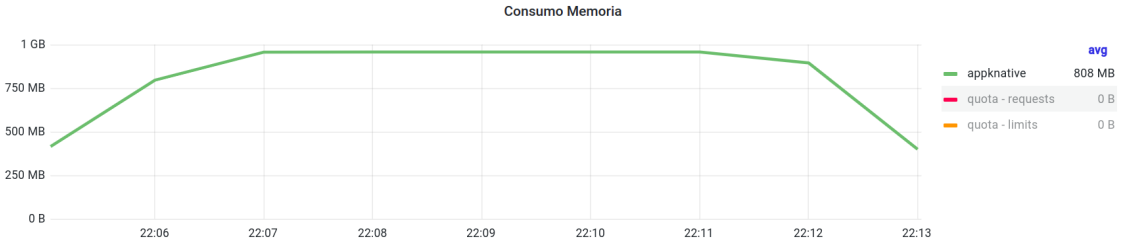



Figura 7.13: Knative service InsertCar Memoria - Risorse utilizzate per l'inserimento singolo

Nel figure sottostanti è possibile vedere come il container *insert-macchine* e il container *queue-proxy* consumino lo stesso quantitativo di CPU mentre nella figura 7.15 viene allocata più memoria al container *insert-car* rispetto al suo sidecar in quanto non avendo richieste da elaborare in coda non necessita di ulteriore memoria.

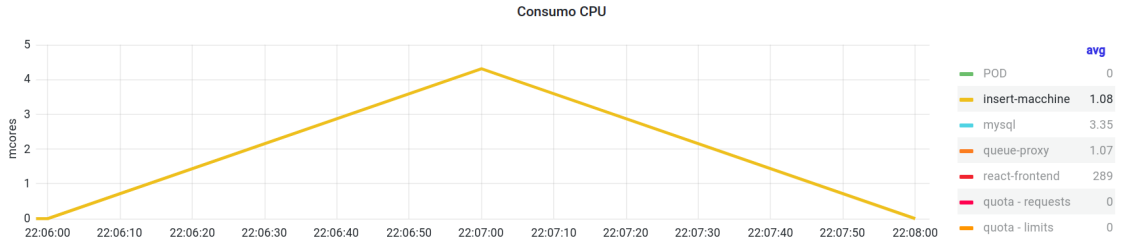



Figura 7.14: Knative service InsertCar Container - Risorse utilizzate per l'inserimento singolo

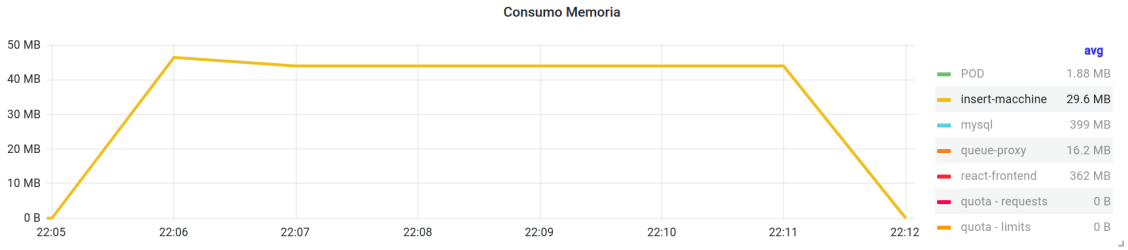



Figura 7.15: Knative Service InsertCar - Memoria singolo inserimento

### 7.2.3 Applicazione AWS

Integrando i servizi di Amazon Web Services, all'interno del cluster viene elaborata la richiesta solo dal *Knative service* incaricato di interfacciarsi con l'utente. Come mostrato in figura 7.16 e 7.17

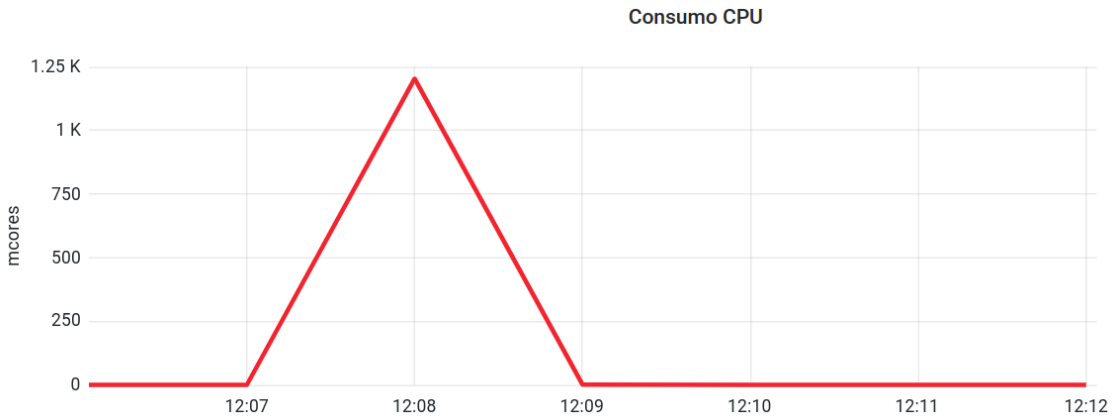



Figura 7.16: Knative Services frontend integrato con AWS - Consumo CPU inserimento singolo

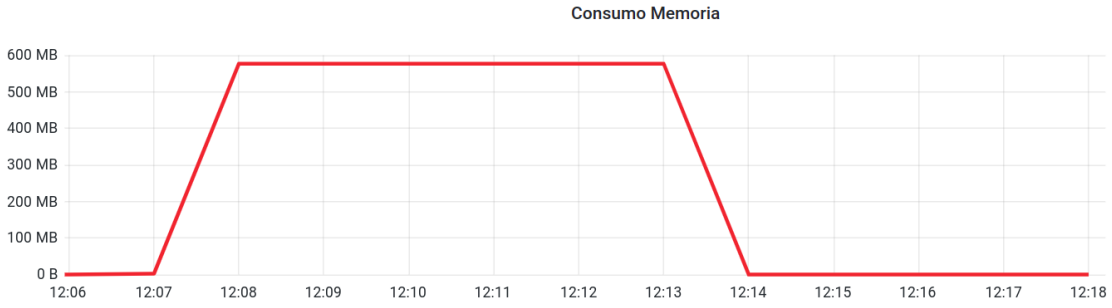



Figura 7.17: Knative Services frontend integrato con AWS - Consumo Memoria inserimento singolo

## 7.3 Inserimento multiplo

L'obiettivo di questo test è valutare la risposta delle applicazioni create in seguito all'inserimento di molteplici dati. Prendendo in considerazione il caso d'uso di un concessionario è possibile ipotizzare che durante l'intero arco di giornata vengano fatte numerosi inserimenti dei clienti all'interno del database. Con questa ipotesi sono state quindi effettuare 5000 richieste nell'arco di 500 secondi in modo tale da poter effettuare uno stress test dell'applicazione.

### 7.3.1 Applicazione monolitica

Osservando il comportamento dell'applicazione durante una fase di stress test è possibile notare nella figura 7.18 come il consumo delle risorse CPU aumenti rispetto allo stato di riposo, con un consumo medio di 54.2 *mcores*.

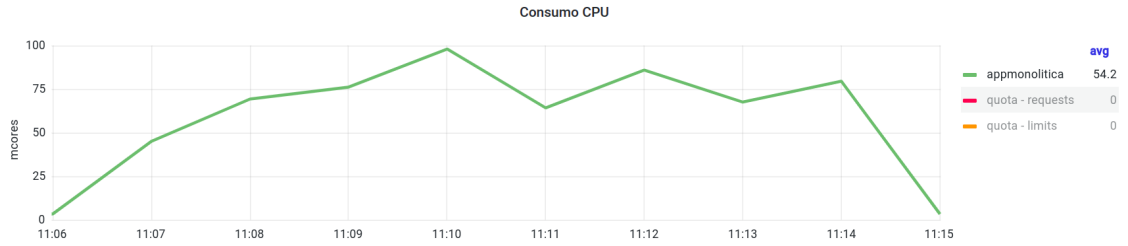



Figura 7.18: Applicazione monolitica InsertClient - Consumo CPU inserimento multiplo

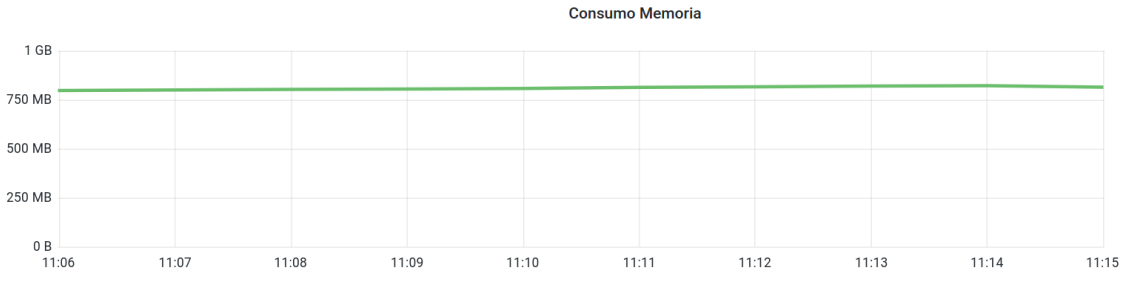



Figura 7.19: Applicazione monolitica InsertClient - Consumo Memoria inserimento multiplo

### 7.3.2 Applicazione serverless

Osservando il comportamento dell'applicazione serverless, l'applicazione nel suo complesso come è possibile notare nella figura 7.20 ha un picco di circa 1000 mcores per poi assestarsi ad un consumo medio di circa 250 *mcores*. Questa situazione è dovuta principalmente alla funzione di autoscaling offerta da knative in quanto ricevendo numerose richieste da parte di *Apache Jmeter* vengono attivati numerosi *Knative Service* per poter soddisfare tale richiesta.

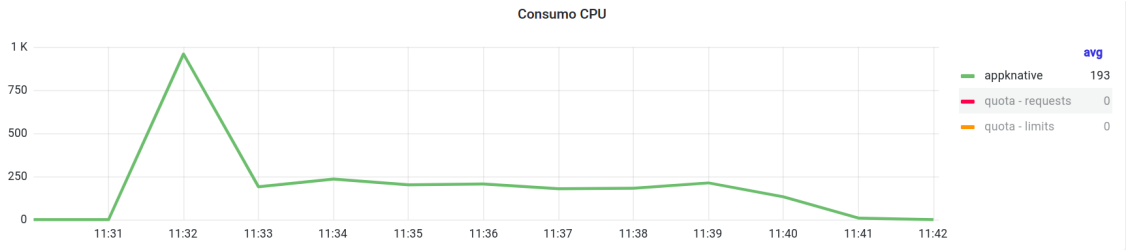



Figura 7.20: Applicazione serverless InsertClient - Consumo CPU inserimento multiplo

Andando ad analizzare più in dettaglio il *Knative Service* responsabile di gestire l'endpoint *POST /insertClient* è possibile notare nella figura 7.21 come il container sidecar *Queue-proxy* risulti significativo con un consumo di quasi il doppio rispetto al picco avuto durante l'inserimento singolo. Lo stesso discorso lato memoria, nella figura **??** è possibile notare un discreto aumento di allocazione della memoria rispetto all'inserimento singolo.

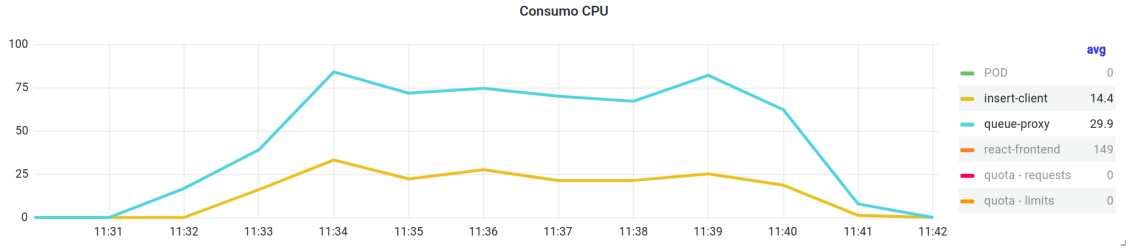



Figura 7.21: Applicazione serverless Container - Consumo CPU inserimento multiplo

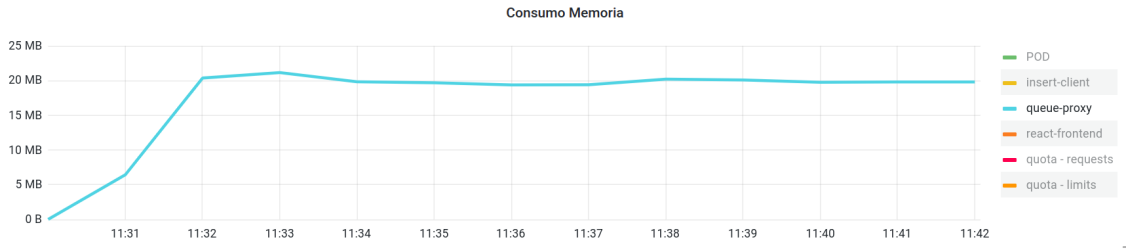



Figura 7.22: Applicazione serverless Queue-Proxy - Consumo Memoria inserimento multiplo

### 7.3.3 Applicazione con AWS

In modo analogo a quello visto con l'inserimento singolo, in questo contesto l'utilizzo di risorse all'interno del cluster viene eseguito solo dal *Knative service* incaricato di fornire l'interfaccia grafica verso l'utente.

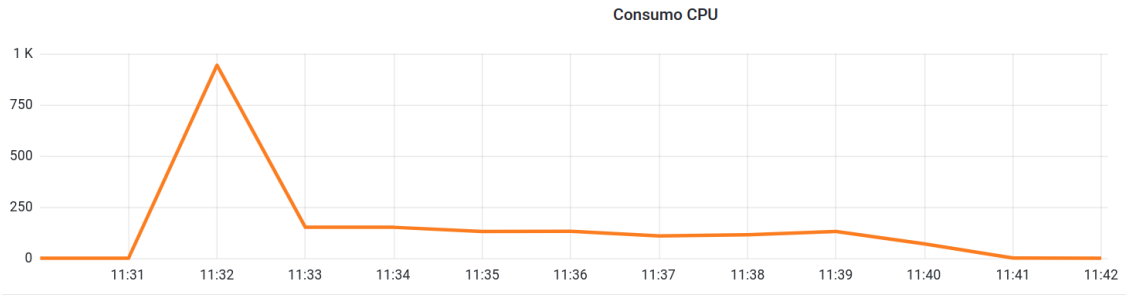



Figura 7.23: Applicazione serverless AWS - Consumo CPU inserimento multiplo

Per quanto riguarda il database implementato all'interno di *Aurora Serverless* è possibile attraverso il servizio *Amazon CloudWatch* vedere quanta CPU viene utilizzata in percentuale, come è possibile vedere nella figura 7.24.


Figura 7.24: Aurora Serverless - Consumo CPU inserimento multiplo

## 7.4 Singola richiesta di dati

In questa sezione si andranno ad eseguire dei test per poter confrontare la risposta delle applicazioni a seguito di una richiesta di una *GET*. Prendendo in considerazione il caso d'uso di un concessionario, è lecito aspettarsi che la richiesta dei dati sui clienti non venga eseguito molto frequentemente, i test pertanto verranno eseguiti sull'endpoint *GET /getClient*.

### 7.4.1 Applicazione monolitica

In modo analogo a quanto visto per l'inserimento singolo, anche in questo caso il consumo di risorse non risulta cambiare in modo significativo rispetto allo stato di riposo visto nelle figure 7.1 e 7.2.

### 7.4.2 Applicazione serverless

Osservando le figure 7.25 e 7.26 è possibile notare come il consumo di CPU e di RAM aumenti in modo significativo nel momento in cui viene invocato il *Knative Service* responsabile di mostrare all'utente i dati relativi ai clienti.

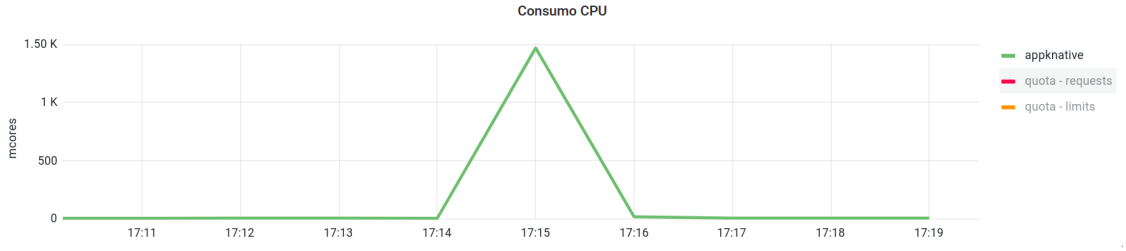



Figura 7.25: Applicazione serverless - Consumo CPU lettura singola

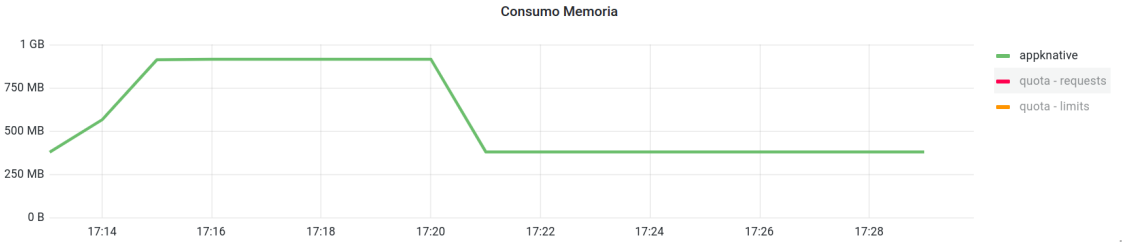



Figura 7.26: Applicazione serverless - Consumo memoria lettura singola

Rispetto all'inserimento singolo, il consumo di CPU mostrato nelle figure 7.27 e r7.28 risulta piuttosto alto per quanto riguarda il *Knative Service* responsabile di raccogliere i dati dal database. La causa è dovuto al fatto che i dati presi attraverso la GET vengono poi elaborati per essere mandati al servizio responsabile di mostrare i dati all'utente

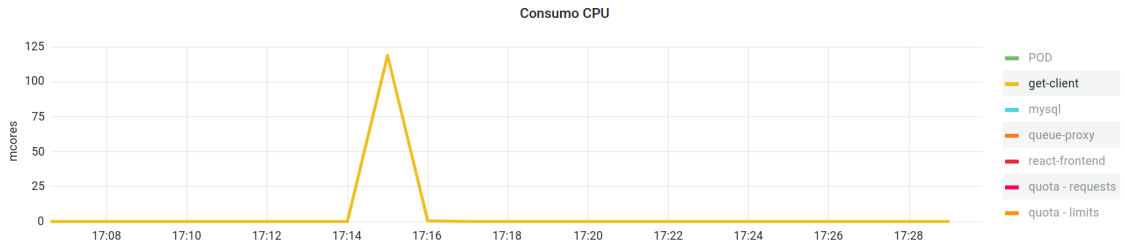



Figura 7.27: Applicazione serverless getClient- Consumo CPU lettura singola

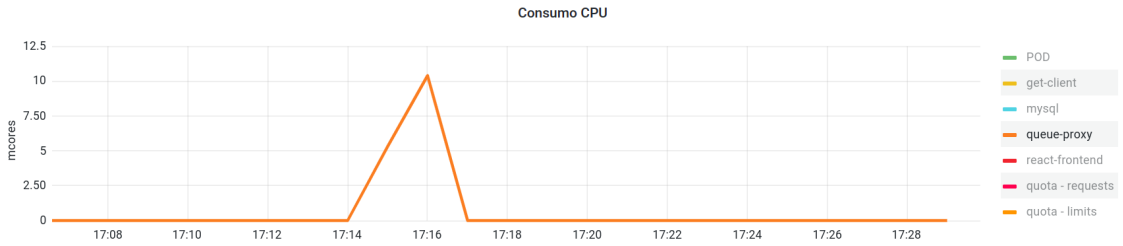



Figura 7.28: Applicazione serverless Queue-proxy - Consumo memoria lettura singola

In modo analogo è possibile vedere come la memoria allocata per rispondere alla GET risulti più alta rispetto al container *Queue-proxy*.

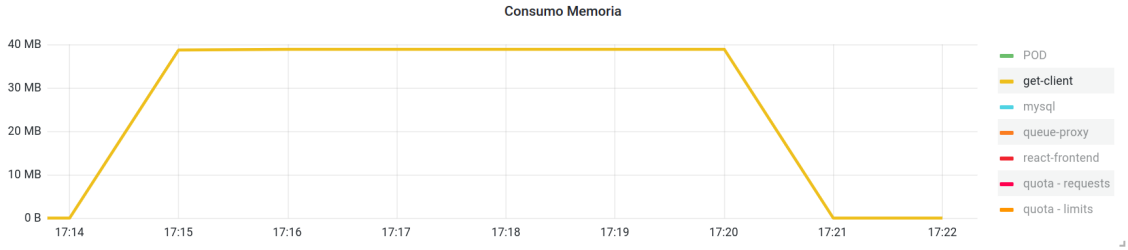



Figura 7.29: Applicazione serverless getClient - Consumo memoria lettura singola

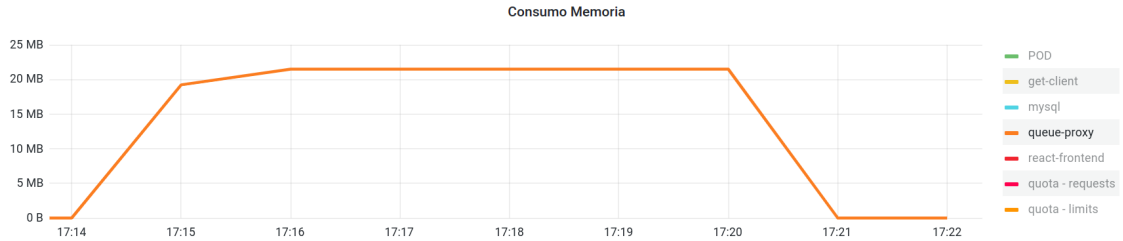



Figura 7.30: Applicazione serverless Queue-proxy - Consumo memoria lettura singola

## 7.5 Richieste multiple di dati

L'obiettivo di questo test è valutare la risposta delle applicazioni create in seguito a numero richieste di dati verso un singolo endpoint. Prendendo in considerazione il caso d'uso di un concessionario è possibile ipotizzare che durante l'intero arco di giornata vengano fatte numerose richieste sulle macchine disponibili. Con questa ipotesi, sono state quindi effettuate 5000 richieste nell'arco di 500 secondi in modo tale da poter effettuare uno stress test dell'applicazione verso l'endpoint *GET /getMacchine*.

### 7.5.1 Applicazione monolitica

Osservando il comportamento dell'applicazione durante la fase di stress test è possibile notare nella figura 7.31 come il consumo di CPU aumento in modo significativo rispetto allo stato di riposo ma anche rispetto alla figura 7.18 a fronte di una elaborazione dei dati ricevuti dal database. Per quanto riguarda l'allocazione di memoria non ci sono stati aumenti significativi.

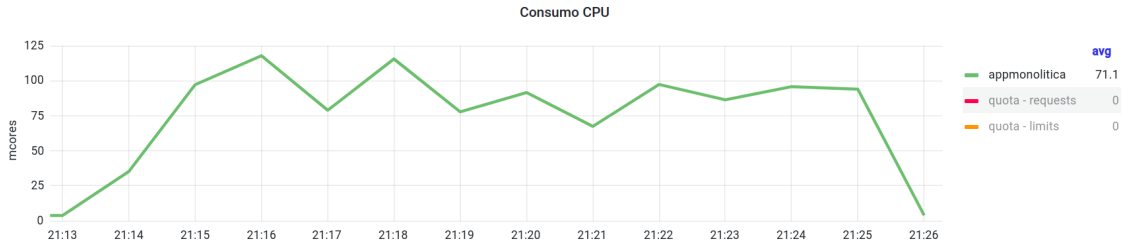



Figura 7.31: Applicazione Monolitica - Consumo CPU lettura multipla

### 7.5.2 Applicazione Serverless

Osservando il comportamento dell'applicazione serverless in figura 7.32 durante la fase di stress test è possibile notare come anche qui si ha un picco all'inizio per poter soddisfare le richieste in arrivo per poi stabilizzarsi verso i 250 *mcores*

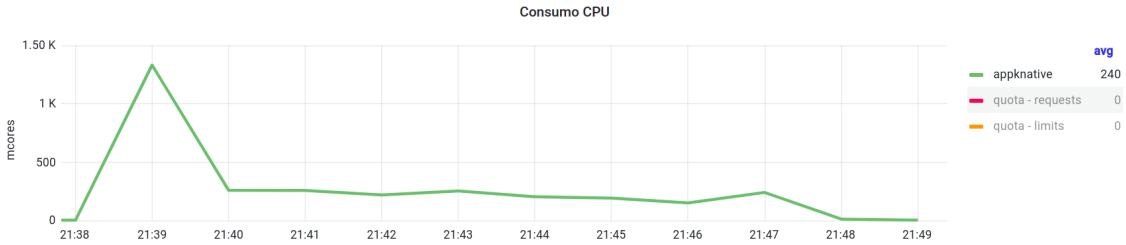



Figura 7.32: Applicazione Serverless - Consumo CPU lettura multipla

Analizzando il *Knative Service* responsabile di rispondere alla richiesta *GET /getMacchine* è possibile notare che rispetto ad una singola richiesta il container *Queue-proxy* ha un consumo più alto.

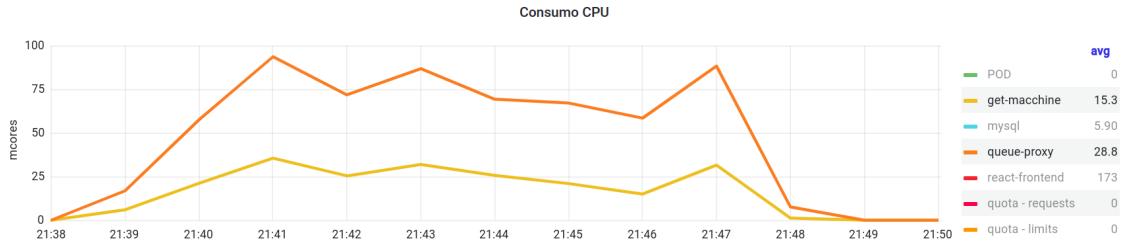



Figura 7.33: Applicazione Serverless Container - Consumo CPU lettura multipla

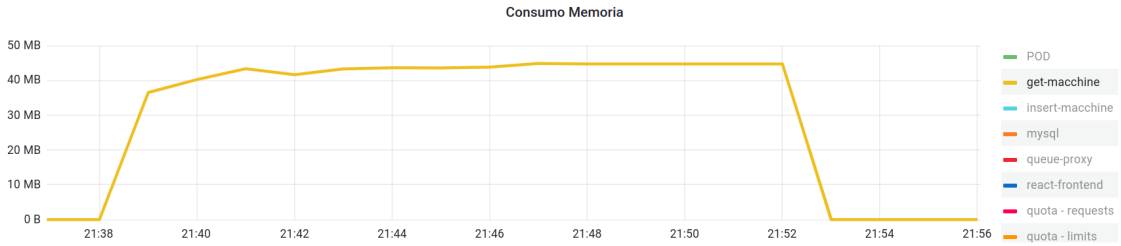



Figura 7.34: Applicazione Serverless getCar - Consumo Memoria lettura multipla

### 7.5.3 Applicazione AWS

In modo analogo all'inserimento multiplo, il consumo di risorse all'interno del cluster è dovuto solo al *Knative Service* responsabile di interfacciarsi con l'utente e, in questo caso, mostrare il catalogo delle macchine. Rispetto all'inserimento multiplo non ci sono cambiamenti significativi sul consumo di risorse, sia per il Knative Service che per Aurora Serverless.

# Capitolo 8

# Analisi e Costi

In questo capitolo si andranno a stimare e confrontare in termini quantitativi le architetture analizzate nel capitolo precedente ipotizzando il caso d'uso di un concessionario. Successivamente, prendendo in considerazione i calcoli svolti si andranno ad analizzare i costi che queste architetture potrebbero influire all'interno di un cluster di *Amazon Web Services* e se l'utilizzo del framework Knative possa risultare una valida alternativa.

## 8.1 Formule utilizzate

Per confrontare le architetture nel modo più oggettivo possibile in termini di CPU e memoria come criterio di calcolo è stata utilizzata la media ponderata:

$$\overline{x} = \frac{\sum_i \overline{x}_\mathrm{i} \cdot \triangle t_\mathrm{i}}{T} \tag{8.1}$$

nel quale $\overline{x}_\mathrm{i}$ indica il valore medio catturato durante i test nella durata $\triangle t_\mathrm{i}$ considerando T l'intero arco della giornata. In generale otteniamo :

$$\overline{CPU} = \frac{\sum_i \overline{CPU}_\mathrm{i} \cdot \triangle t_\mathrm{i}}{T} \tag{8.2}$$

$$\overline{Memoria} = \frac{\sum_i \overline{memoria}_\mathrm{i} \cdot \triangle t_\mathrm{i}}{T} \tag{8.3}$$

I valori medi di $\overline{cpu}_\mathrm{i}$ e $\overline{memoria}_\mathrm{i}$ utilizzati nelle formule sono stati presenti durante la fase di test del capitolo precedente. Per ragioni pratiche i test effettuati sull'applicazione sono stati eseguite su un tempo limitato, i calcoli che andremo ad effettuare si è scelto di utilizzare comunque i valori medi dei test ipotizzando che le risorse impiegate risultino simili anche per un periodo di tempo prolungato.

## 8.2 Inserimento

In questa sezione si andranno ad analizzare i consumi delle risorse mostrati nel capitolo precedente sull'inserimento di dati all'interno dell'applicazione.

### 8.2.1 Inserimento Singolo

In questo scenario, nel capitolo precedente si è ipotizzato che l'inserimento di macchine venga fatto in modo saltuario, pertanto si considereranno due fasce orarie,ciascuna della durata di un ora, nel quale verranno eseguite 4 chiamate ad ogni fascia oraria. I valori medi di questi dati sono presi considerando solo i *Knative Services* senza considerare il container incaricato di contenere il database.In quanto l'obiettivo è valutare le performance di *Knative* senza comprendere quelle di *MySQL*. In particolare otteniamo :

$$\overline{CPU}_{\text{monoltica}} = 3.5\, mcores \tag{8.4}$$

$$\overline{Memoria}_{\text{monolitica}} = 800\, MB \tag{8.5}$$

$$\overline{CPU}_{\text{FaaS}} = 294\, mcores \tag{8.6}$$

$$\overline{Memoria}_{\text{FaaS}} = 407\, MB \tag{8.7}$$

Nel caso di Knative è possibile vedere nel figura 7.12 come il servizio resta attivo per soli 2 minuti, pertanto ipotizzando due fasce orarie nel quale viene chiamata la risorsa si ottiene:

$$\overline{CPU}_{\text{monolitica}} = \frac{3.5\, mcores \cdot 2\, h}{24\, h} = 0.58\, mcores \tag{8.8}$$

$$\overline{Memoria}_{\text{monolitica}} = \frac{800\, MB \cdot 2\, h}{24\, h} = 66.6\, MB \tag{8.9}$$

$$\overline{CPU}_{\text{FaaS}} = \frac{294\, mcores \cdot 0.066\, h}{24\, h} = 0.8085\, mcores \tag{8.10}$$

$$\overline{Memoria}_{\text{FaaS}} = \frac{407\, MB \cdot 0.66\, h}{24\, h} = 1.11\, MB \tag{8.11}$$

### 8.2.2 Inserimento Multiplo

In questa sezione viene ipotizzato l'utilizzo dell'endpoint */insertClient* nell'arco di 4 fasce orarie, ciascuna della durata di un'ora, durante l'intero arco della giornata. In modo

analogo all'inserimento singolo, i valori medi dell'applicazione FaaS sono stati presi senza considerare il container *MySQL*.

$$\overline{CPU}_{\text{monoltica}} = 54.2\, mcores \tag{8.12}$$

$$\overline{Memoria}_{\text{monolitica}} = 800\, MB \tag{8.13}$$

$$\overline{CPU}_{\text{FaaS}} = 193\, mcores \tag{8.14}$$

$$\overline{Memoria}_{\text{FaaS}} = 192\, MB \tag{8.15}$$

Ponendo T = 4 ore possiamo utilizzare le formule 8.2 e 8.3 per poter stimare l'utilizzo giornaliero delle risorse impiegate:

$$\overline{CPU}_{\text{monolitica}} = \frac{54.2\, mcores \cdot 4\, h}{24\, h} = 9.03\, mcores \tag{8.16}$$

$$\overline{Memoria}_{\text{monolitica}} = \frac{800\, MB \cdot 4\, h}{24\, h} = 133.34\, MB \tag{8.17}$$

$$\overline{CPU}_{\text{FaaS}} = \frac{193\, mcores \cdot 4\, h}{24\, h} = 32.16\, mcores \tag{8.18}$$

$$\overline{Memoria}_{\text{FaaS}} = \frac{192\, MB \cdot 4\, h}{24\, h} = 32\, MB \tag{8.19}$$

## 8.3 Lettura dati

In questa sezione in modo analogo alla sezione precedente, si andrà ad analizzare il comportamento dell'applicazione durante l'intero arco della giornata, ipotizzando delle richieste di lettura di dati.

### 8.3.1 Lettura Singola

Nel capitolo precedente si è ipotizzato che la risorsa */getClient* all'interno di una giornata tipo venga chiamata raramente. Nel nostro scenario si ipotizza dunque che nell'arco di due fasce orarie, ciascuna di un'ora, vengono effettuate 4 chiamate in ogni fascia orario. In modo analogo all'inserimento dei dati nel nostro contesto i valori medi dell'applicazione eseguita su *Knative* vanno considerati senza il container di *MySQL*. I valori medi sono:

$$\overline{CPU}_{\text{monoltica}} = 3.5\, mcores \tag{8.20}$$

$$\overline{Memoria}_{\text{monolitica}} = 800\, MB \tag{8.21}$$

$$\overline{CPU}_{\text{FaaS}} = 450\, mcores \tag{8.22}$$

$$\overline{Memoria}_{\text{FaaS}} = 424\, MB \tag{8.23}$$

Nel caso di Knative è possibile vedere nel figura 7.12 come il servizio resta attivo per soli 2 minuti, pertanto ipotizzando due fasce orarie nel quale viene chiamata la risorsa si ottiene:

$$\overline{CPU}_{\text{FaaS}} = \frac{450\, mcores \cdot 0.066\, h}{24\, h} = 1.23\, mcores \tag{8.24}$$

$$\overline{Memoria}_{\text{FaaS}} = \frac{424\, MB \cdot 0.066\, h}{24\, h} = 1.16\, MB \tag{8.25}$$

I risultati dell'applicazione monolitica si possono vedere nelle equazioni 8.8 e 8.9 in quanto i valori risultano uguali.

### 8.3.2 Lettura multipla

Nel capitolo precedente si è ipotizzato che la risorsa */getCar* all'interno di giornata tipo venga chiamata molto frequentemente. Nel nostro scenario si ipotizza dunque che nell'arco di 4 fasce orarie, ciascuna dalla durata di un'ora, vengano effettuate numerose chiamate verso l'endpoint */getCar*.In modo analogo ai calcoli precedenti nel nostro contesto i valori medi dell'applicazione eseguita su *Knative* vanno considerati senza il container di *MySQL*. I valori medi ottenuti sono:

$$\overline{CPU}_{\text{monoltica}} = 71.1\, mcores \tag{8.26}$$

$$\overline{Memoria}_{\text{monolitica}} = 800\, MB \tag{8.27}$$

$$\overline{CPU}_{\text{FaaS}} = 240\, mcores \tag{8.28}$$

$$\overline{Memoria}_{\text{FaaS}} = 440\, MB \tag{8.29}$$

Ponendo T = 4 ore le risorse utilizzate durante l'intero arco della giornata sono :

$$\overline{CPU}_{\text{monolitica}} = \frac{71.1\, mcores \cdot 4\, h}{24\, h} = 11.85\, mcores \tag{8.30}$$

$$\overline{Memoria}_{\text{monolitica}} = \frac{800\, MB \cdot 4\, h}{24\, h} = 133.34\, MB \tag{8.31}$$

$$\overline{CPU}_{\text{FaaS}} = \frac{240\,mcores \cdot 4\,h}{24\,h} = 40\,mcores \tag{8.32}$$

$$\overline{Memoria}_{\text{FaaS}} = \frac{440\,MB \cdot 4\,h}{24\,h} = 73.3\,MB \tag{8.33}$$

## 8.4 Knative

Nel capitolo precedente sono stati presi i grafici anche per quanto riguardo il consumo delle risorse di Knative, difatti questo framework è consuma discrete risorse hardware che devono essere considerate nell'intero calcolo.

$$\overline{CPU} = 44.3\,mcores \tag{8.34}$$

$$\overline{Memoria} = 205\,MB \tag{8.35}$$

Ponendo T = 8 ore ovvero un intero arco di giornata lavorativa otteniamo i seguenti risultati:

$$\overline{CPU} = \frac{44.3\,mcores \cdot 8\,h}{24\,h} = 14.7\,mcores \tag{8.36}$$

$$\overline{Memoria} = \frac{205\,MB \cdot 8\,h}{24\,h} = 68.33\,MB \tag{8.37}$$

## 8.5 Scenario giornaliero

Ottenuti i dati degli endpoint sul quale sono stati effettuati i test risulta necessario combinare i vari elementi ottenuti al fine di avere una stima giornaliera. Nella tabella 8.1 vengono riassunti i dati raccolti nelle sezioni precedenti. Combinando i dati otteniamo :

| **Richieste** | **Applicazione Monolitica** | | **Applicazione Knative** | |
|---|---|---|---|---|
| | **CPU** | **Memoria** | **CPU** | **Memoria** |
| **Inserimento Cliente** | 9.03 mcores | 133.34 MB | 32.16 mcores | 32 MB |
| **Inserimento Macchina** | 0.58 mcores | 66.6 MB | 0.8085 mcores | 1.11 MB |
| **Richiesta Clienti** | 9.03 mcores | 133.34 MB | 1.23 mcores | 1.16 MB |
| **Richiesta Macchine** | 11.85 mcores | 133.34 MB | 40 mcores | 73.3 MB |
| **Knative(a riposo)** | | | 14.7 mcores | 68.3 MB |

Tabella 8.1: Tabella riassuntiva dei dati raccolti

$$\overline{CPU}_{\text{Monolitica}} = (9.03 + 0.58 + 9.03 + 11.85) = 30.50 mcores \tag{8.38}$$

$$\overline{Memoria}_{\text{Monolitica}} = (133.34 + 66.6 + 133.34 + 133.34) = 466.62MB \tag{8.39}$$

$$\overline{CPU}_{\text{FaaS}} = (32.16 + 0.8085 + 1.23 + 40) = 74.19mcores \tag{8.40}$$

$$\overline{Memoria}_{\text{FaaS}} = (32 + 1.11 + 1.16 + 73.3 + 68.3) = 175.87MB \tag{8.41}$$

Per quanto riguarda l'applicazione Knative, nel momento in cui all'interno di un cluster viene utilizzato solo per far funzionare questa applicazione allora dovranno essere aggiunti nel conto:

$$\overline{CPU}_{\text{FaaS+Knative}} = 74.19 + 14.7 = 88.90mcores \tag{8.42}$$

$$\overline{Memoria}_{\text{FaaS+Knative}} = 176.87 + 68.3 = 245.17MB \tag{8.43}$$

La tabella 8.5 mostra i risultati ottenuti in questi calcoli, misurando l'utilizzo di CPU in mcores e di memoria in MB.

| | **Applicazione Monolitica** | | **Applicazione Knative** | |
|---|---|---|---|---|
| | **CPU** | **Memoria** | **CPU** | **Memoria** |
| **Totale Giornaliero** | 30.50 mcores | 466.62 MB | 74.19 mcores | 175.87 MB |
| **Knative** | | | 88.90 mcores | 245.17 MB |

Tabella 8.2: Tabella risorse utilizzate giornalmente

Da questi risultati, risulta evidente come l'architettura serverless offerta da Knative ,con le considerazioni effettuate, permetta un notevole risparmio lato memoria, con un consumo ridotto del 62% ma con un aumento del 58% di consumo lato CPU. Questa discrepanza lato memoria è dovuta principalmente agli endpoint che vengono utilizzati poche volte rispetto alla parte monolitica che rimane sempre attiva. Risulta evidente quindi come il tenere attivo un *Knative Service* in modo continuo senza sfruttare la sua capacità di poter scalare a zero aumentino le risorse computazionali.

## 8.6 Analisi Costi

In questa sezione si andrà ad analizzare in termini di costi le varie architetture messe in gioco. Come è stato ampiamente detto in questa tesi il cloud provider utilizzato è *Amazon Web Services* che permette di poter istanziare un proprio cluster in cloud permettendo notevoli risparmi. In generale l'utilizzo di un cluster in cloud rispetto ad una soluzione

on-premise permette un notevole risparmio offrendo una maggiore stabilità, flessibilità ed efficienza.

### 8.6.1 Condizioni Generali

Le applicazioni sono state inserite all'interno di un cluster EKS, vedere sezione 5.2, al suo interno sono state istanziate due macchine EC2 on-demand di tipo t3.medium i quali hanno rispettivamente 4 CPU e 4 GB di memoria. La scelta di questo tipo di istanza deriva principalmente da poter permettere a Knative di funzionare in modo ottimale in quanto per funzionare richiede minimo 2 CPU e 4 GB di memoria. Inoltre all'interno del cluster di Amazon è stato implementato un **Load Balancer** ed un **Elastic IP**. L'utilizzo di EKS permette una facile gestione di *Kubernetes* in quanto attraverso un semplice click è possibile aggiornare ed eliminare qualsiasi nodo, permettendo di fatto un notevole risparmio in termini di tempo rispetto ad una soluzione on-premises. In generale i costi di un'implementazione attraverso questi servizi sono :

EKS = 0.10 $ ogni ora
EC2 = 0,0416 $ ogni ora
Load Balancer = 0.027 $ ogni ora

Inoltre si evidenza come all'interno di ogni macchina EC2 si può avere un certo numero limitato di pod al suo interno.Questa limitazione è dovuta alla network interface dell'istanza in quanto per ogni POD viene assegnato un indirizzo IP pubblico. In questo contesto ogni istanza EC2 t3.medium permette di contenere al suo interno 17 pod.

### 8.6.2 Applicazione Monolitica

In questo contesto, utilizzando solo Kubernetes come piattaforma nel quale deployare le proprie applicazioni, il pod seppur in modo minimo consuma delle risorse all'interno di un'istanza EC2. In generale quindi per un tempo T= 24 ore:

$$Giornaliero_{\text{Monolitica}} = 0.10 \cdot T + 0.0416 \cdot T + 0.027 \cdot T = 4\,\$ \tag{8.44}$$

$$Mensile_{\text{Monolitica}} = 121\,\$ \tag{8.45}$$

Per una verifica oggettiva sui reali costi si ipotizza che all'interno di un cluster EKS vengono gestite più applicazioni. Ipotizzando un'azienda che vuole gestire 100 pod di cui il 90% solo durante la giornata lavorativa il numero di macchine EC2 salirà in quanto ogni istanza EC2 è limitata dal numero di indirizzi IP che possono essere assegnati. In questo caso quindi il numero di macchine EC2 sale fino a 6 rispetto ad una sola macchina ipotizzata per una sola applicazione.

$$Giornaliero_{\text{Monolitica}\cdot 100} = 0.10 \cdot T + 0.0416 \cdot T \cdot 6 + 0.027 \cdot T = 9\,\$ \tag{8.46}$$

$$Mensile_{\text{Monolitica}\cdot 100} = 271\,\$ \tag{8.47}$$

### 8.6.3 Applicazione Knative

In questo contesto, il framework Knative essendo on-top di Kubernetes utilizza delle risorse per poter funzionare in modo ottimale. L'utilizzo di Knative fa in modo che una macchina EC2 con le caratteristiche iniziali venga dedicata per il suo funzionamento in quanto i pod installati da Knative più quelli di Kubernetes risultano 14 a fronte dei 17 disponibili. Con il fine di stimare il costo di un'applicazione con Knative che potesse funzionare in modo ottimale, si è deciso di instanziare un'ulteriore macchine EC2 t3.medium. Nonostante Knative grazie all'autoscaling a zero permetterebbe di non pagare i costi della seconda macchina EC2 quando non viene utilizzata si è ipotizzato che l'applicazione venga continuamente utilizzata solo all'interno di una giornata lavorativa.

$$Giornaliero_{\text{Knative}} = 0.10 \cdot T + 0.0416 \cdot T + 0.0416 \cdot 8h + 0.027 \cdot T = 13.20\,\$ \tag{8.48}$$

$$Mensile_{\text{Knative}} = 396{,}192\,\$ \tag{8.49}$$

Per una verifica più realistica dei costi, si ipotizza di avere 100 Pod creati attraverso *Knative* utilizzati per il 90% durante una giornata lavorativa. In modo analogo al caso precedente anche in questo contesto il numero di macchine EC2 salirà, in questo caso a 7.

$$Giornaliero_{\text{Knative}\cdot 100} = 0.10 \cdot T + 0.0416 \cdot T \cdot 2 + 0.0416 \cdot 8h \cdot 5 + 0.027 \cdot T = 6.7\,\$ \tag{8.50}$$

$$Mensile_{\text{Knative}\cdot 100} = 201\,\$ \tag{8.51}$$

### 8.6.4 Applicazione con AWS

In questo contesto oltre ai servizi utilizzati nelle altre architetture si sfruttano ulteriori servizi di Amazon. In particolare vengono utilizzati *AWS Lambda* e *Aurora Serverless* . Prendendo in considerazione la nostra applicazione avremo un solo pod creato con Knative per poi spostare tutta la parte backend su Amazon. I costi dei servizi sono i seguenti :

$$\text{AWS Lambda} = 0.20 \quad \$ \text{ per un milione di richieste}$$
$$\text{AWS Aurora} = 0.096 \ \$ \text{ ogni ora per un'istanza di tipo db.t3.medium}$$

Ipotizzando che la nostra applicazione venga utilizzata solo durante una giornata di 8 ore le richieste di Lambda risultano molto inferiori rispetto al milione per questo motivo nel calcolo giornaliero non è stato inserito il suo costo.

$$Giornaliero_{\text{AWS}} = 0.10 + 0.0416 \cdot T + 0.20 + 0.096 \cdot 8 + 0.027 \cdot T = 4.90\,\$ \tag{8.52}$$

$$Mensile_{\text{AWS}} = 150.4\,\$ \tag{8.53}$$

Per una verifica più realistica dei costi all'interno di Amazon, si è ipotizzato anche qui di utilizzare 100 Pod realizzati con Knative di cui il 90 % viene utilizzato solo durante la giornata lavorativa.

$$Giornaliero_{\text{Knative}\cdot 100} = 0.10 \cdot T + 0.0416 \cdot T \cdot 2 + 0.0416 \cdot 8h \cdot 5 + 0.027 \cdot T + 0.20 + 0.096 \cdot 8h = 7.2\$ \tag{8.54}$$

$$Mensile_{\text{AWS}\cdot 100} = 217.3\,\$ \tag{8.55}$$

## 8.7 Raccolta dati

Ottenuti i dati sui prezzi è possibile eseguire un confronto tra le diverse soluzioni proposte. Come è possibile notare dalla tabella 8.3 il prezzo per mantenere una singola applicazione all'interno di un cluster va a favore dell'applicazione monolitica con un risparmio del 69% rispetto a Knative e del 19% rispetto all'utilizzo dei servizi di Amazon.

| | **Applicazione Monoltica** | **Applicazione Knative** | **Applicazione con AWS** |
|---|---|---|---|
| **Giornaliero** | 4 $ | 13.20 $ | 4.90 $ |
| **Mensile** | 121 $ | 396.192 $ | 150.4 $ |

Tabella 8.3: Tabella Prezzi singola applicazione

Naturalmente il discorso cambia radicalmente inserendo più servizi all'interno del proprio cluster infatti come è possibile vedere dal grafico 8.1 più i servizi aumentano più l'utilizzo di solo Kubernetes diventa alto. Per quanto riguarda Knative e la sua integrazione con i servizi Amazon il costo risulta molto simile, inoltre c'è da considerare che nei calcoli si ipotizza il caso peggiore dove vengono utilizzati tutti i servizi per tutta la giornata lavorativa. Nel momento in cui non vengono utilizzati i prezzi di entrambe le soluzioni andrebbero a diminuire grazie alla possibilità di far scalare a zero i Pod e di un costo dell'utilizzo basato a ore.

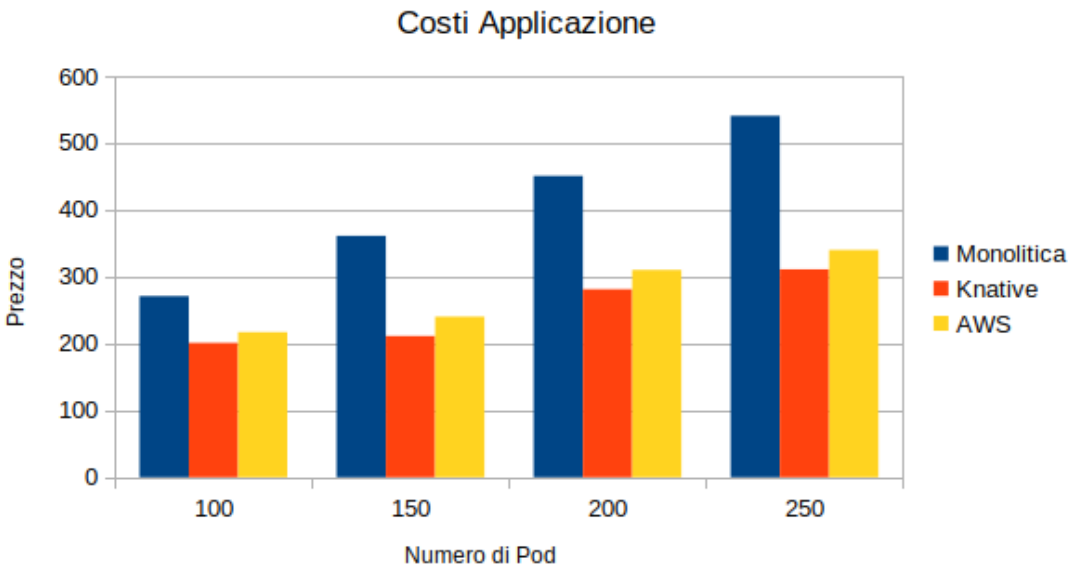



Figura 8.1: Grafico Prezzi

# Capitolo 9

# Conclusioni finali

Il lavoro di tesi ha portato all'analisi del framework Knative e dei servizi offerti da AWS. Si è potuto notare nel capitolo precedente come l'obiettivo è stato raggiunto, in particolare è possibile notare due aspetti.

- L'applicazione realizzata con Knative permette di poter avere un risparmio dei costi a carico delle aziende nel caso in cui si utilizzano numerosi servizi. Se, da una parte, otteniamo un risparmio in termini di costi, dall'altro lato si avrà un aumento significativo delle risorse hardware utilizzate nel momento in cui il servizio verrà attivato.
- L'applicazione realizzata integrando Knative con i servizi offerti da AWS permette un risparmio significativo rispetto all'applicazione monolitica ma non quanto la controparte Knative. É da notare però come l'avere i servizi di AWS permettono di avere notevoli risparmi in termini di tempo e personale per la gestione dei servizi all'interno di un cluster. Ad esempio, per database di grande dimensioni, la sicurezza e la gestione completamente automatica da parte di AWS potrebbe essere un valore aggiunto che possa giustificare un aumento dei costi.

In generale Knative risulta essere un framework capace di far abbassare i costi alle aziende, risulta però non idoneo per quei servizi che vengono utilizzati molto frequentemente e che richiedono una risposta veloce.

# Bibliografia


[1] Amid Khatibi Bardsiri eyyed Mohsen Hashemi. « Cloud computing vs Grid computing». In: (2012).

[2] Timothy Grance Peter M. Mell. « The NIST Definition of Cloud Computing ». In: (2011), p. 2. URL: https://www.nist.gov/publications/nist-definition-cloud-computing.

[3] Bitnami Project. *Kubeless*. URL: https://kubeless.io/.

[4] Openfaas. *Introduction to OpenFaas*. URL: https://docs.openfaas.com/.

[5] Docker inc. *Docker*. URL: https://www.docker.com/.

[6] RedHat. «What is docker». In: (2018). URL: https://www.redhat.com/it/topics/containers/what-is-docker.

[7] Mohammad Ahmadi Babak Bashari Rad Harrison John Bhatti. «An Introduction to Docker and Analysis of its Performance». In: (2017).

[8] Kubernetes. *What is kubernetes*. URL: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/.

[9] Kubernetes. *Understanding Kubernetes Objects*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/.

[10] Luksa Marko. *Kubernetes in Action*. A cura di Manning Pubns Co. 2nd edition. 2018, pp. 143, 160.

[11] Kubernetes. *Viewing Pods and Nodes*. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/.

[12] Red Hat. *What is Knative?* URL: https://www.redhat.com/it/topics/microservices/what-is-knative.

[13] CNCF. *Survey Report*. URL: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[14] Istio. URL: https://istio.io/.

[15] Ambassador. URL: https://www.getambassador.io/.

[16] Contour. URL: https://projectcontour.io/.

[17] Kourier. URL: https://github.com/knative-sandbox/net-kourier.

[18] Knative. *Knative Serving Autoscaling System*. URL: https://github.com/knative/serving/blob/main/docs/scaling/SYSTEM.md.

[19] Knative. URL: https://knative.dev/docs/eventing/.

[20] Amazon Web Services. URL: https://aws.amazon.com/it/.

[21] Amazon Web Services. *AWS Region*. URL: https://aws.amazon.com/it/about-aws/global-infrastructure/regions_az/.

[22] Amazon Web Services. *AWS EC2*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html.

[23] Amazon Web Services. *AWS EKS*. URL: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html.

[24] Amazon Web Services. *AWS Lambda*. URL: https://aws.amazon.com/it/lambda/.

[25] Amazon Web Services. *AWS RDS*. URL: https://aws.amazon.com/it/rds/.

[26] Amazon Web Services. *AWS Aurora*. URL: https://aws.amazon.com/it/rds/aurora/?aurora-whats-new.sort-by=item.additionalFields.postDateTime&aurora-whats-new.sort-order=desc.

[27] Bref. URL: https://bref.sh/.

[28] Wordpress. URL: https://wordpress.com/it/.

[29] React. URL: https://it.reactjs.org/.

[30] Prometheus. URL: https://prometheus.io/.

[31] Grafana. URL: https://grafana.com/.